MASTER EN SCIENCES INFORMATIQUES

Optimisation des performances de programmation Java par caching

Charles, Philippe

Award date:
2007

Awarding institution:
Universite de Namur

Facultés Universitaires Notre-Dame de le Paix, Namur
Institut d'informatique
Année académique 2006-2007

Optimisation des performances de programmation Java par caching

Philippe Charles

Mémoire présenté en vue de l'obtention du grade de licencié en informatique

Résumé

Il existe plusieurs techniques pour optimiser les performances d'un programme, comme l'optimisation algorithmique ou encore l'optimisation mémoire. Ce travail porte sur l'étude de l'optimisation par effet de cache, appelée caching.

Mots clés

cache, caching, memoization, optimisation, performance, efficacité, java, jcache, jsr-107, éviction, méta-programmation, annotations, PAO, profilage, caching dynamique

Abstract

There are several techniques to optimize the performances of an application, such as algorithmic optimization or memory optimization. The goal of this thesis is to study the optimization done with a cache, also called caching.

Keywords

cache, caching, memoization, optimization, performance, efficiencyn java, jcache, jsr-107, eviction, metaprogramming, annotations, POA, profiling, dynamic caching

Remerciements

Je tiens à remercier mon promoteur, Monsieur Wim Vanhoof, professeur aux Facultés Universitaires Notre-Dame de La Paix à Namur, et son assistant, Monsieur Stéphane Bonfitto, pour leurs conseils et la supervision de ce travail.

Je veux exprimer plus particulièrement ma gratitude à mon chef de service, Monsieur Jean Palate, et à la Banque Nationale de Belgique, qui m'ont permis d'associer les recherches effectuées dans le cadre de ce mémoire à une application concrète en cours de développement.

Mes chaleureux remerciements vont enfin à mes collègues Jérémy Demortier et Sébastien Leroy, ainsi qu'à tous ceux qui m'ont soutenu dans cette entreprise par leurs avis, relectures et encouragements.

TABLE DES MATIÈRES

TABLE DES FIGURES

LISTE DES TABLEAUX

LISTINGS

GLOSSAIRE

Complexité Estimation de l'efficacité des algorithmes.

Java Specification Request (JSR) Document formel qui décrit des propositions de spécifications et technologies à ajouter à la plateforme Java.

Least Recently Used (LRU) Algorithme de gestion de mémoire qui élimine l'élément le moins récemment utilisé.

Not Frequently Used (NFU) Algorithme de gestion de mémoire qui élimine l'élémént le moins fréquemment utilisé depuis son chargement.

Pattern En génie informatique, un motif de conception (design pattern en anglais) est un concept issu de la programmation orientée objet, destiné à résoudre les problèmes récurrents. En français on utilise aussi le synonyme patron de conception.

Profilage Le profilage (profiling en anglais) consiste à analyser les performances d'un programme lors de son exécution.

Singleton Dans le domaine des design patterns, un singleton correspond à une classe dont il n'existe qu'une seule instance.

Time-to-live (TTL) Le Time-to-live (temps de vie) indique le temps pendant lequel une information doit être conservée.

Transtypage Conversion de type en orienté objet, aussi appelé coercition de type ou encore casting.

Wiki Un wiki est un site web dynamique permettant à tout individu d'en modifier les pages à volonté.

INTRODUCTION

L'optimisation des performances par utilisation de caching est une technique courante de l'informatique. On la retrouve partout, que ce soit du côté matériel avec par exemple les différents niveaux de cache d'un processeur, ou du côté logiciel comme le cache d'un navigateur web.

Cette solution logicielle n'est pourtant pas bien maîtrisée par les développeurs et est souvent mal employée. En effet, sa relative simplicité masque en réalité une grande complexité d'implémentation et d'utilisation qui rend sa manipulation difficile.

Parmi les problèmes courants rencontrés par les programmeurs, on peut citer l'altération du code métier par celui de gestion du cache, l'utilisation d'une solution de caching incomplète ou trop ciblée, la gestion délicate de la cohérence des données, la faible publicité du caching, ou encore la planification et l'intégration fastidieuses.

Ce mémoire porte donc sur l'étude de l'optimisation des performances par caching, Java servant à illustrer de manière pratique les solutions envisagées. Le mémoire se divise en trois parties.

La première vise à définir clairement le concept de caching et à répertorier les différentes fonctionnalités offertes.

La deuxième porte sur les différentes techniques d'intégration du caching dans le processus de développement. Elle sert d'une part, à identifier les parties de programme susceptibles d'être optimisées et d'autre part, à fournir une méthodologie concrète et efficace pour l'incorporer au sein d'une application.

Enfin, la troisième et dernière partie se propose de mettre en pratique les concepts abordés dans les deux premières parties, l'expérience ainsi acquise permettant d'éprouver la théorie face à la dure réalité.

Première partie

Principes généraux

CHAPITRE 1

PRÉSENTATION DU CACHING

L'optimisation des performances est un problème récurrent en informatique. Bien que les ordinateurs modernes soient de plus en plus puissants, ceux-ci restent toujours limités d'une façon ou d'une autre et tout programmeur est vite confronté à ces limites.

Il existe plusieurs techniques différentes pour améliorer l'efficacité d'un programme, comme l'optimisation algorithmique ou encore l'optimisation mémoire. Nous nous intéresserons ici à l'optimisation par effet de cache, appelée caching.

Prenons d'abord un exemple pour illustrer celui-ci.

Dans le livre « Le Guide du Voyageur galactique »[Adams, 1979], une race d'extra-terrestres construit un super ordinateur dont le but est de calculer la réponse à la grande question sur la vie, l'univers et le reste. Une fois cet ordinateur construit, ils lui posent la fameuse question. Cette dernière étant fort complexe, la machine met plusieurs millions d'années pour trouver la réponse[1].

A la fin du calcul, si cette machine n'enregistrait pas quelque part la réponse, elle serait obligée de la recalculer à chaque fois que quelqu'un la lui poserait au lieu de la donner directement. L'enregistrement de la réponse pour une utilisation ultérieure se fait par l'utilisation d'un cache.

[1] La réponse est bien évidemment 42.

Concrètement, un cache est une zone de stockage contenant des ressources ou des résultats de traitements dont le coût de production initial (souvent exprimé en temps d'accès) est plus important que leur lecture dans le cache. Une fois que des données sont copiées dans un cache, elles peuvent être accédées plus rapidement que si elles devaient être recalculées ou récupérées.

Les caches sont utilisés dans de nombreux domaines.

Ils peuvent servir d'interface entre deux médias dont les débits et temps d'accès sont différents comme, par exemple, un navigateur web et un site web distant. Dans ce cas, un disque dur local sera utilisé pour stocker des parties du site qui ne changent pas souvent pour en optimiser l'affichage dans le navigateur.

Ils peuvent aussi contenir les résultats de traitements coûteux comme des requêtes imposantes dans des bases de données ou des calculs scientifiques complexes. Une fois l'information produite, elle peut être diffusée à tous les processus qui en ont ou en auront besoin.

Prenons comme exemple le calcul de la suite de Fibonacci. L'algorithme de cette suite peut être écrit naïvement de cette façon :

```
public class Fibonacci {
  public int fibonacci(int n) {
    if (n < 2)
      return n;
    return (fibonacci(n - 1) + fibonacci(n - 2));
  }
}
```

Listing 1.1: Fibonacci

On constate dans ce code l'existence d'appels récursifs recalculant souvent les mêmes cas. Sa complexité est exponentielle ($O(1.6^n)$) mais peut être sérieusement réduite en enregistrant les résultats intermédiaires. En utilisant un tableau d'entiers de taille `max` comme cache, l'algorithme devient :

```
public class Fibonacci {
  private static final int max = 100;

  private static int[] fibo;

  static {
    fibo = new int[max];
    Arrays.fill(fibo, -1);
  }

  public int fibonacci(int n) {
    if (fibo[n] == -1) {
      if (n < 2)
        fibo[n] = n;
      else
```

```
            fibo[n] = (fibonacci(n - 1) + fibonacci(n - 2));
        }
        return fibo[n];
    }
}
```

Listing 1.2: Fibonacci optimisé

La complexité est alors linéaire ($O(n)$). Il serait possible d'optimiser le système en stockant directement dans le tableau les valeurs connues de n pour 0 et 1 lors de son initialisation, mais cela n'a pas été réalisé ici pour ne pas trop altérer l'algorithme initial.

La technique de caching utilisée dans cet algorithme est parfois appelée *memoization*. Selon wikipedia, la *memoization* est « une technique utilisée pour accélérer les traitements de programmes en stockant les résultats des fonctions pour une réutilisation ultérieure plutôt que de devoir les recalculer »[wikipedia.org, 2005]. Il s'agit en fait d'une variante simplifiée de la programmation dynamique [Schobbens, 2002].

La *memoization* est un cas particulier du caching car elle se limite aux opérations dont le résultat ne change pas ou peu lorsque l'on leur passe les mêmes arguments. En général, les résultats de la *memoization* sont pérennes.

Plus généralement, le caching manipule des données susceptibles d'expirer. Leur validité est donc temporaire, ce qui peut amener des problèmes de cohérence de cache.

Si le principe du caching semble simple au premier abord, sa mise en oeuvre est autrement plus compliquée. Le principal problème vient de l'absence d'un standard dans ce domaine. Les programmeurs sont laissés à leur imagination. Il en résulte des portions de code maladroits et/ou ultra-spécialisés qui sont difficilement réutilisables et surtout maintenables par de tierces personnes.

Les chapitres qui suivent serviront à clarifier les concepts du caching et à en extraire une certaine méthodologie.

CHAPITRE 2

FONCTIONNALITÉS ESSENTIELLES

Ce chapitre a pour vocation de dégager les fonctions principales d'un système de caching par la construction d'une solution de base en Java, de manière itérative. Chaque problème ou besoin rencontré sera réglé par l'ajout de fonctionnalités dans le code.

Toutefois, la solution présentée dans les sections suivantes n'a d'autre but que d'illuster les concepts de base du caching. Il existe bien évidemment d'autres solutions plus intéressantes ou mieux adaptées à certains problèmes[1].

2.1 Solution générique

Tout d'abord, une solution de caching doit être suffisament générique pour être réutilisable dans d'autres circonstances. Celle proposée dans le précédent chapitre pour la suite de Fibonacci (listing 1.2) est dans ce cas très mauvaise car liée au problème qu'elle veut résoudre. Une solution générique ne doit pas connaître à l'avance le type d'objets qu'elle va stocker.

Pour ce faire, on va créer une interface dont les méthodes se chargeront d'ajouter et de récupérer des objets. L'avantage d'une interface par rapport à

[1] Pour une liste non exhaustive de solutions open-source, veuillez consulter le site http://www.java-source.net/open-source/cache-solutions

une classe est que l'implémentation de celle-ci est laissée au programmeur qui peut alors choisir la plus adaptée à son problème.

La technique la plus souvent utilisée pour ajouter et récupérer des objets consiste en l'utilisation d'une paire clé-valeur. Un objet est retrouvé dans le cache sur base d'une clé qui le définit. Seules deux méthodes simples sont requises pour gérer cela :

```
public interface Cache {
  void put(Object key, Object value);

  Object get(Object key);
}
```

Listing 2.1: Cache générique

La première méthode, `put`, ajoute un objet dans le cache et lui associe une clé. La seconde méthode, `get`, renvoie un objet sur base de sa clé.

Il suffit alors d'implémenter une classe concrète de l'interface `Cache`, ici `CacheImpl`, et de l'utiliser comme suit dans le cas de l'algorithme de la suite de Fibonacci :

```
public class Fibonacci {
  private static Cache fibo = new CacheImpl();

  public int fibonacci(int n) {
    Integer result = (Integer) fibo.get(n);
    if (result == null) {
      if (n < 2)
        result = n;
      else
        result = fibonacci(n - 1) + fibonacci(n - 2);
      fibo.put(n, result);
    }
    return result.intValue();
  }

}
```

Listing 2.2: Fibonacci optimisé par cache générique

Dans l'exemple ci-dessus, la clé utilisée est l'entier `n` représentant la nième valeur de la suite de Fibonacci. Cet entier est automatiquement transtypé en `Integer`[2] par le compilateur ce qui fait qu'il dérive bien de la classe de base `Object`[3].

Une telle solution de caching peut être facilement implémentée par une `HashMap`[4]. Elles partagent d'ailleurs les méthodes `put` et `get`.

[2] http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Integer.html
[3] http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Object.html
[4] http://java.sun.com/j2se/1.5.0/docs/api/java/util/HashMap.html

On remarque aussi que l'ajout de la solution de caching dans le code original implique une modification plus ou moins importante de celui-ci. Le caractère intrusif des solutions de caching sera discuté dans un chapitre ultérieur.

2.2 Durée de vie des objets cachés

Les objets que l'on stocke dans le cache ne sont pas forcément valides indéfiniment. On pourrait, par exemple, décider que les nouvelles affichées sur une page web doivent être renouvelées toutes les heures. Dans ce cas, il nous faut introduire un nouveau concept qui est la durée de vie des objets (Time To Live, TTL) sans oublier que certains objets peuvent eux avoir une durée de vie infinie, comme les résultats de la suite de Fibonacci.

Pour tenir compte du temps de vie d'un objet dans le cache, il convient de dédoubler la méthode `put` introduite précédemment en faisant appel aux capacités de polymorphisme de Java.

```
public interface Cache {
  void put(Object key, Object value);

  void put(Object key, Object value, long ttl);

  Object get(Object key);
}
```

Listing 2.3: Cache générique avec ttl

La nouvelle méthode `put` dispose d'un nouvel argument, `ttl`, qui définit le temps de vie en millisecondes de l'objet dans le cache. L'appel de la première méthode donne à l'objet un temps de vie infini.

Pour gérer le `ttl`, il est également nécessaire d'ajouter une nouvelle interface qui décrit l'état de l'objet dans le cache.

```
public interface CacheEntry {
  long getCreationTime();

  long getExpirationTime();

  boolean isValid();
}
```

Listing 2.4: Meta-données de chaque objet du cache

Chaque nouvel objet ajouté dans le cache se voit associer un `CacheEntry` qui détermine si l'objet est toujours valide au moment où on le demande. La méthode la plus utilisée, `isValid()`, renvoie `true` si l'objet contenu dans le

cache est toujours valide, `false` dans le cas contraire. Une implémentation possible de cette méthode pourrait se baser sur la date d'expiration de l'objet et la date courante.

2.3 Gestion de la mémoire

La solution proposée jusqu'à présent recèle pourtant un gros défaut. Elle fait l'impasse sur la gestion de la mémoire. En effet, la mémoire d'un ordinateur n'est pas infinie. On ne peut pas stocker des objets sans limite. Ce problème est implicitement abordé dans l'optimisation de l'algorithme de la suite de Fibonacci par l'utilisation d'un tableau borné (listing 1.2), la limite de stockage étant fixée par la constante `max`.

La gestion de la mémoire n'est pas facile et dépend souvent du cas d'utilisation et du contexte. Dans la gestion du cache d'un navigateur web, par exemple, la limite est définie par la taille de l'espace disque que l'on lui a réservé. Une fois cet espace plein, le navigateur web doit commencer à faire de la place pour ajouter les nouveaux fichiers qu'il désire conserver. Ceci passe par la suppression des fichiers les plus anciens et/ou les moins utilisés.

Dans le cadre de la programmation pure, cette gestion est similaire. Le cache doit faire de la place quand l'espace qui lui est alloué arrive à saturation. Les objets qui sont supprimés du cache sont choisis sur base de critères. On appelle cela la *stratégie d'éviction*.

C'est le cache qui détermine quand il a besoin de faire de la place. A l'opposé, le choix des objets à supprimer est laissé à une classe externe. Cela permet d'utiliser différentes stratégies d'éviction avec un même cache en fonction du contexte de l'application.

Pour prendre en compte la gestion de la mémoire, il faut modifier le cache générique comme suit :

```
public interface Cache {
  void put(Object key, Object valeur);

  void put(Object key, Object valeur, long ttl);

  Object get(Object key);

  Object remove(Object key);

  void clear();

  void evict();
}
```

Listing 2.5: Cache générique avec gestion memoire

Les méthodes `remove` et `clear` permettent au programmeur de nettoyer manuellement le cache en supprimant un objet bien précis ou en vidant complètement le cache. De même, la méthode `evict` lance un nettoyage automatique basé sur la stratégie d'éviction courante.

Pour qu'une stratégie d'éviction soit opérationnelle, elle a besoin d'un certain nombre d'informations qui lui sont fournies par le `CacheEntry` au moyen des méthodes suivantes :

```
public interface CacheEntry {
  long getCreationTime();

  long getExpirationTime();

  boolean isValid();

  int getHits();

  long getLastAccessTime();

  long getLastUpdateTime();

  Object getValue();

  Object setValue(Object value);
}
```

Listing 2.6: Meta-données de chaque objet du cache orientées stratégie d'éviction

La méthode `getHits`, par exemple, renvoie le nombre de fois que l'objet a été consulté dans le cache. Elle permet de classer les objets par fréquence d'utilisation (Not Frequently Used, NFU). Les méthodes `getLastAccessTime` et `getLastUpdateTime` permettent quant à elles de créer une stratégie d'éviction basée sur les derniers objets utilisés (Least Recently Used, LRU).

Enfin, pour que ces nouvelles méthodes puissent fonctionner, il faut que le `CacheEntry` gère lui-même l'accès aux objets cachés. Les méthodes `getValue` et `setValue` jouent ce rôle. Le `CacheEntry` n'est plus alors un simple descripteur de données mais aussi un accesseur.

La stratégie d'éviction est pour sa part représentée par l'interface `EvictionStrategy`. Celle-ci dispose d'une méthode `evict` qui renvoie une `map`[5] de clés d'objets à supprimer du cache.

```
public interface EvictionStrategy {

  public void clear();

  public CacheEntry createEntry(Object key, Object value, long ttl);
```

[5] http://java.sun.com/j2se/1.5.0/docs/api/java/util/Map.html

```
  public void discardEntry(CacheEntry entry);
  public Map evict(Cache cache);
  public void touchEntry(CacheEntry entry);
}
```

Listing 2.7: Stratégie d'éviction

Les autres méthodes présentes dans cette interface servent à informer la stratégie d'éviction des événements survenus dans le cache.

CHAPITRE 3

FONCTIONNALITÉS AVANCÉES

Si un système de cache se doit de posséder un certain nombre de fonctionnalités basiques et indispensables, il n'en demeure pas moins qu'il existe nombre de services supplémentaires qu'il peut offrir aux programmeurs pour leur faciliter la tâche.

Ce chapitre va décrire les principaux services que l'on attend d'un cache. Ceux-ci ne sont en aucun cas indispensables et peuvent être utilisés ou pas en fonction des besoins.

De plus, il faut savoir que chaque service ajouté à la solution de base apporte un coût de traitement supplémentaire et s'avère donc susceptible de pénaliser les performances, ce qui est contre-productif. Il est donc nécessaire de ne choisir que les fonctionnalités vraiment utiles, au cas par cas si possible.

3.1 Gestion des exceptions

La gestion des exceptions tient plus de la nécessité que de la fonctionnalité. En effet, les services qui sont décrits dans ce chapitre sont succeptibles de lever des exceptions suite à des erreurs d'exécution ou de programmation.

La classe décrite ci-dessous se chargera de propager les exceptions liées au cache.

```
public class CacheException extends Exception {

  public CacheException() {
    super();
  }

  public CacheException(String s) {
    super(s);
  }

  public CacheException(String s, Throwable ex) {
    super(s, ex);
  }
}
```

Listing 3.1: Gestion des exceptions

3.2 Gestionnaire de cache

Il n'est pas rare de devoir utiliser plusieurs caches dans un programme. Chacun peut avoir des caractéristiques propres ou simplement regrouper des objets d'un type commun.

Un gestionnaire de cache est une classe particulière qui se charge de gérer l'ensembe des caches et d'y donner accès à travers l'application. Il permet aussi de réduire le code nécessaire à l'utilisation d'un cache, celui-ci étant instancié et configuré en dehors du code métier.

Un gestionnaire de cache est un singleton doté d'une première méthode permettant de renvoyer un cache sur base de son nom et d'une deuxième méthode permettant d'ajouter un nouveau cache dans le gestionnaire. Son implémentation est la suivante :

```
public class CacheManager {
  private static CacheManager singleton = new CacheManager();

  public static CacheManager getInstance() {
    return singleton;
  }

  private HashMap map;

  private CacheManager() {
    map = new HashMap();
  }

  public Cache getCache(String name) {
    return (Cache) map.get(name);
  }

  public void registerCache(String cacheName, Cache cache) {
```

```
    map.put(cacheName, cache);
  }
}
```

Listing 3.2: Gestionnaire de cache

Son diagramme de séquence est la figure 3.1 à la page 14. Ce diagramme montre la création d'un cache par un client, son enregistrement dans le gestionnaire de cache et enfin sa récupération ultérieure par un autre client.

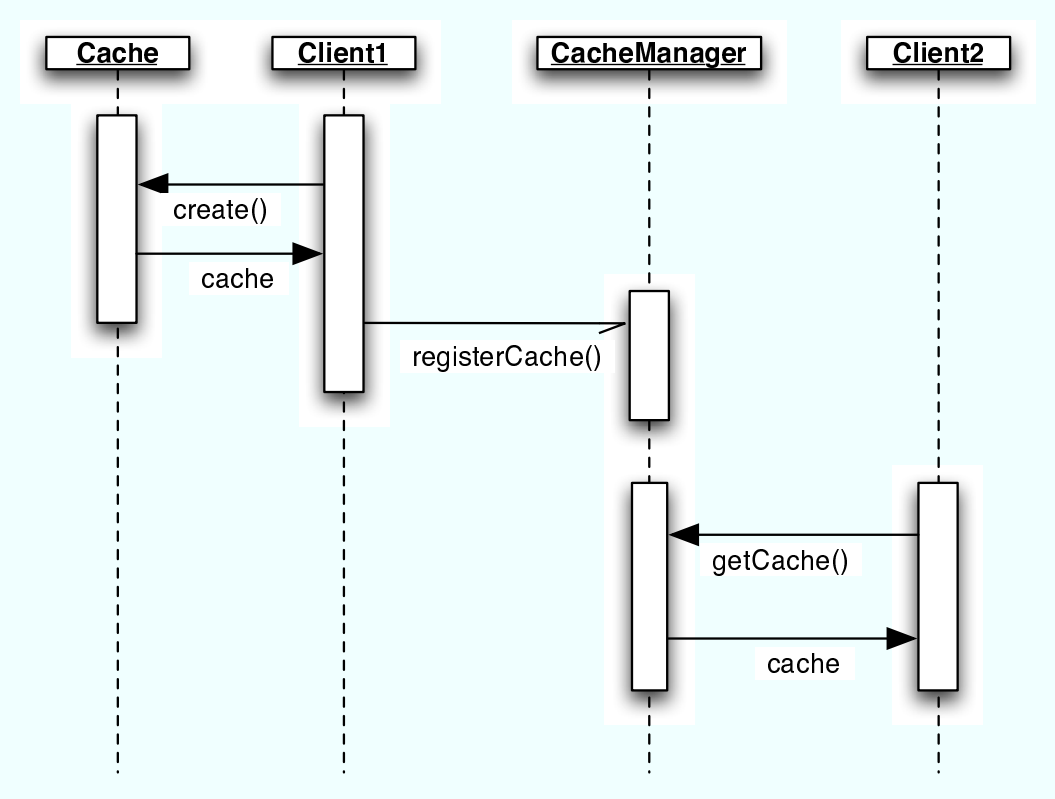

FIG. 3.1: Diagramme de séquence d'un gestionnaire de cache

Son utilisation dans l'exemple de la suite de Fibonacci (listing 2.2) est fort simple et ne modifie pas le code de la fonction :

```
public class Fibonacci {

  static {
    CacheManager.getInstance().registerCache("fibonacci",
        new CacheImpl());
  }

  static Cache fibo = CacheManager.getInstance().getCache(
      "fibonacci");
```

```
  int fibonacci(int n) {
    Integer result = (Integer) fibo.get(n);
    if (result == null) {
      if (n < 2)
        result = n;
      else
        result = fibonacci(n - 1) + fibonacci(n - 2);
      fibo.put(n, result);
    }
    return result.intValue();
  }
}
```

Listing 3.3: Fibonacci avec gestionnaire de cache

L'un des avantages les plus intéressant d'un gestionnaire de cache est l'externalisation de la création et de la configuration du cache. En effet, la mémoire disponible pour une application peut varier en fonction de la machine sur laquelle elle fonctionne. Il est possible qu'une machine de développement soit moins bien équipée qu'un serveur en production. Dans ce cas, les paramètres du cache doivent être ajoutés à l'exécution et non codés en dur dans la méthode cachée.

3.3 Fabrique abstraite de cache

Si l'on observe le gestionnaire de cache introduit précédemment, on s'apperçoit que, même s'il est possible de créer un cache en dehors de la classe qui l'utilise, il est toujours nécessaire d'appeler une classe concrète connue à la compilation. Le choix de la classe est alors figé dans le code.

Pour éviter ce genre de choses, on a recours au pattern de fabrique abstraite. Une fabrique abstraite est un motif de conception créationnel qui encapsule un groupe de fabriques particulières ayant une thématique commune. Le code client crée une implémentation concrète de la fabrique abstraite, puis utilise les interfaces génériques pour créer des objets concrets de la thématique. Le client ne se préoccupe pas de savoir quels objets concrets il obtient de chacune des fabriques, car il n'utilise que les interfaces génériques de leurs produits. Ce motif de conception sépare les détails d'implémentation d'un ensemble d'objets de leur usage générique. [Gamma *et al.*, 1998].

La fabrique abstraite de cache est donc une interface qui suit ce pattern et sert à créer un nouveau cache sur base de paramètres d'environnement. Cette classe est implémentée par les fournisseurs de systèmes de cache.

C'est le gestionnaire de cache qui est chargé de renvoyer la fabrique courante grâce à sa méthode `getCacheFactory`. Le choix de la fabrique est réalisé sur base de la configuration du gestionnaire. Ce mécanisme est représenté par la figure 3.2 à la page 17. Son code est le suivant :

```
public class CacheManager {
  protected static CacheManager instance = new CacheManager();

  public static CacheManager getInstance() {
    return instance;
  }

  private HashMap map;

  private CacheManager() {
    map = new HashMap();
  }

  public Cache getCache(String name) {
    return (Cache) map.get(name);
  }

  public void registerCache(String cacheName, Cache cache) {
    map.put(cacheName, cache);
  }

  public CacheFactory getCacheFactory() throws CacheException {
    String className = System
        .getProperty("CacheFactory.ClassName");
    try {
      return (CacheFactory) Class.forName(className)
          .newInstance();
    } catch (Exception e) {
      throw new CacheException(e.getMessage(), e);
    }
  }

}
```

Listing 3.4: Gestionnaire de cache avec fabrique abstraite

```
public interface CacheFactory {
  Cache createCache(Map env) throws CacheException;
}
```

Listing 3.5: Fabrique abstraite de cache

La méthode `getCacheFactory` est susceptible de lancer une exception dans le cas où le gestionnaire de cache serait incapable de renvoyer une fabrique, ce qui peut arriver lorsque la configuration est erronée.

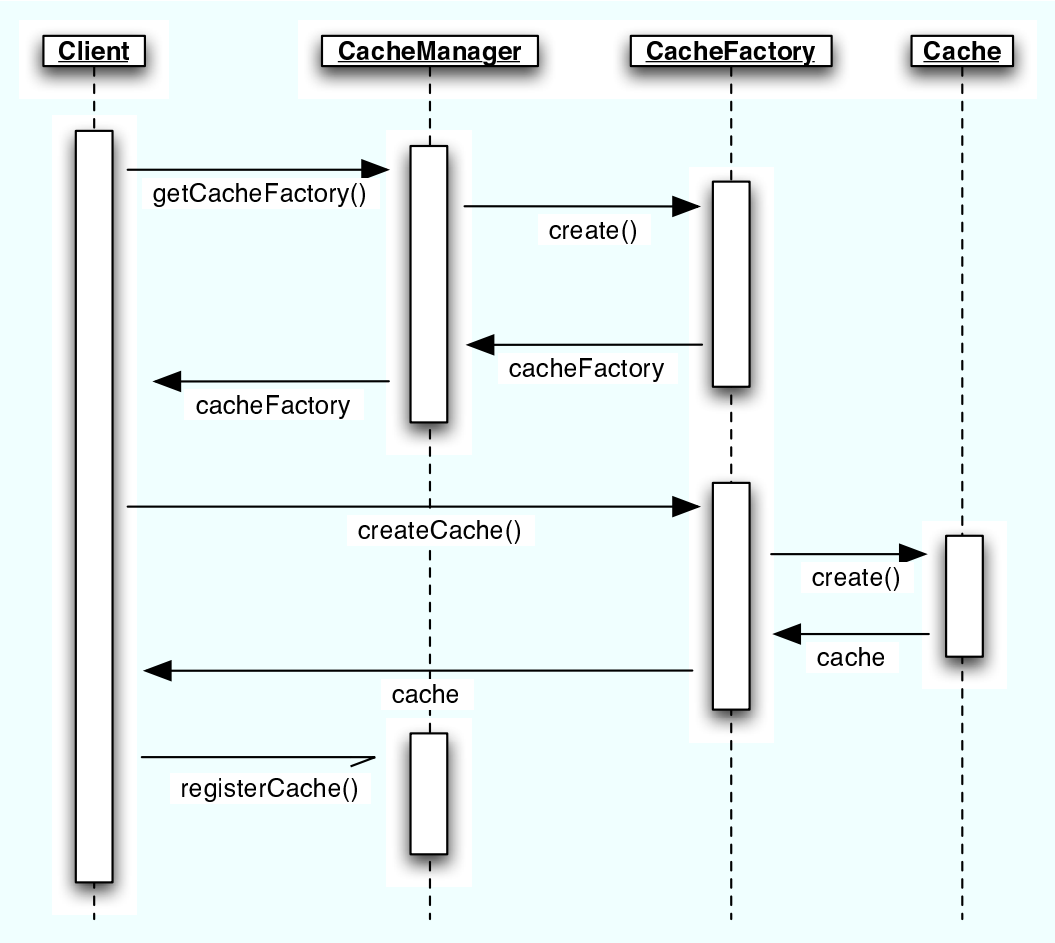

FIG. 3.2: Diagramme de séquence d'un gestionnaire de cache avec fabrique abstraite

3.4 Cache sur plusieurs niveaux

A l'image de la gestion de la mémoire d'un ordinateur par un OS, il est imaginable que le cache soit implémenté sur plusieurs niveaux. Un premier niveau, le plus rapide, garde les objets en mémoire vive. Un deuxième niveau stocke les objets sur un média plus lent, comme le disque dur, mais disposant de plus d'espace.

Dans ce but, un cache a besoin d'un système de chargement qui récupère des objets qui ne seraient pas disponibles dans le cache. Ce système, appelé `CacheLoader`, dispose de deux méthodes ; la première, récupère un seul objet, tandis que la deuxième renvoie un ensemble d'objets. Son fonctionnement

est représenté par la figure 3.3 à la page 18.

```
public interface CacheLoader {

  public Object load(Object key) throws CacheException;

  public Map loadAll(Collection keys) throws CacheException;

}
```

Listing 3.6: Loader de cache

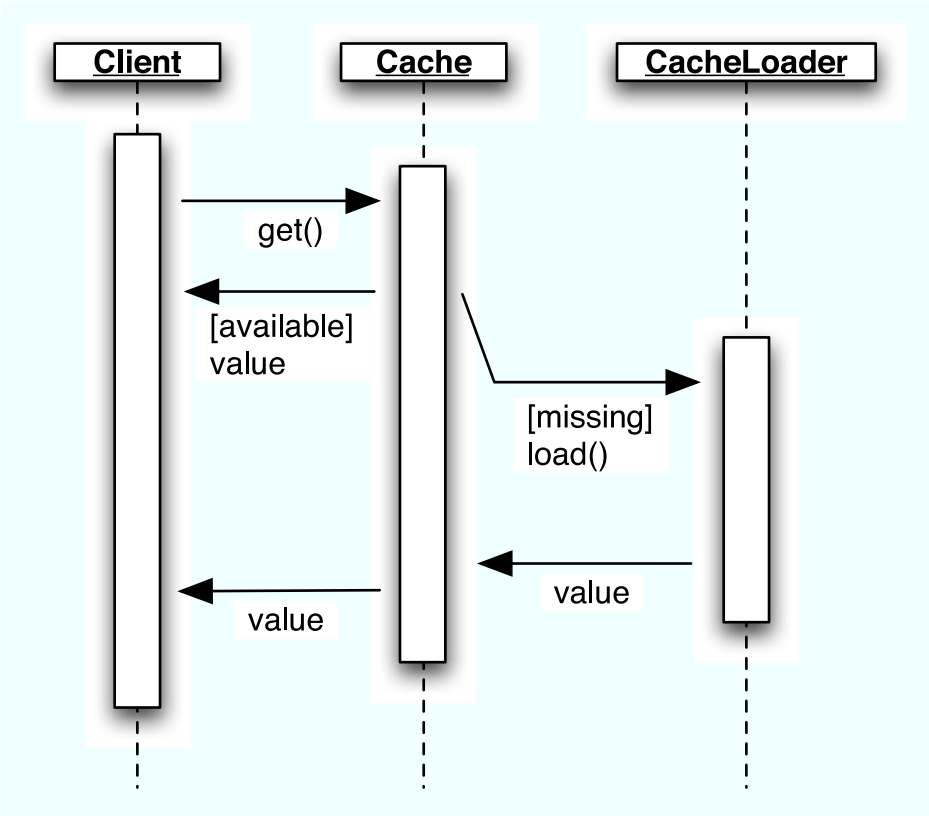

FIG. 3.3: Diagramme de séquence d'un cache sur plusieurs niveaux

Un chargement venant d'un niveau inférieur amène vraisemblablement un délai supplémentaire dans la récupération d'un objet.

3.5 Audit des performances

L'utilisation d'un cache entraine un coût non négligeable du fait des différents traitements qu'il effectue pour retrouver un objet. Il n'est donc pas

toujours judicieux d'utiliser un cache dans du code. Le coût d'utilisation du cache doit être inférieur au coût de production des données.

De plus, les performances d'un cache dépendent souvent de sa configuration. Par exemple, plus on alloue d'espace mémoire au cache, moins il devra avoir recours à sa stratégie d'éviction, ce qui implique moins de traitements pour l'éviction et moins d'appels au code business pour la création des objets manquants.

Il faut donc être en mesure de jauger l'efficacité d'un cache. Ceci peut être réalisé au moyen de quelques statistiques simples en enregistrant par exemple le nombre de fois que le cache a ou n'a pas trouvé un objet, ainsi que le nombre d'objets présents dans le cache.

L'interface de gestion de statistiques est la suivante :

```
public interface CacheStatistics {

  public void clearStatistics();

  public int getCacheHits();

  public int getCacheMisses();

  public int getObjectCount();

}
```

Listing 3.7: Audit des performances

En outre, la méthode `clearStatistics` remet les compteurs de `hit` et `miss` à zéro.

Bien évidemment, ces statistiques sont assez basiques, pour ne pas dire limitées. Dans le cadre d'une application critique, il sera opportun d'avoir recours à un outil spécialisé dans le profilage [Java-Source.net, 2006b].

3.6 Multi-threading

Dans un environnement moderne, peu de programmes sont mono-thread. Par exemple, les interfaces graphiques font intervenir plusieurs threads ; un pour l'affichage et un pour le calcul. Ce genre de choses permet d'offrir à l'utilisateur plus d'interactivité et de réactivité. L'avenir des processeurs est d'ailleurs multi-coeurs. Dans ce cadre, la programmation par thread devient une nécessité si l'on veut profiter de la puissance mise à disposition.

Or, la programmation par thread apporte son lot de problèmes. D'une part, par sa complexité à scinder les tâches, d'autre part, par des problèmes de synchronisation, d'exclusion mutuelle (deadlock) et d'accès concurrents.

Si l'on veut rendre un cache accessible à plusieurs threads en même temps, il faut alors ajouter au code d'implémentation du cache des instructions de bloquage. Ce genre d'instructions est assez mal aisé à mettre en oeuvre si on veut éviter des bugs et autres deadlocks.

La programmation d'un cache thread-safe est envisageable de trois façons.

La première méthode copie la classe d'implémentation de `Cache` dans une nouvelle classe et ajoute le code de bloquage.

```
public class ThreadSafeCache implements Cache {

  synchronized public void put(Object key, Object valeur) {
    // TODO Auto-generated method stub
  }

  synchronized public void put(Object key, Object valeur, long ttl) {
    // TODO Auto-generated method stub
  }

  synchronized public Object get(Object key) {
    // TODO Auto-generated method stub
    return null;
  }

  synchronized public Object remove(Object key) {
    // TODO Auto-generated method stub
    return null;
  }

  synchronized public void clear() {
    // TODO Auto-generated method stub
  }

  synchronized public void evict() {
    // TODO Auto-generated method stub
  }

}
```

Listing 3.8: Cache thread-safe par copie

Cette méthode a comme gros désavantage d'obliger le programmeur à maintenir en parallèle deux classes quasiment identiques. Par contre, c'est celle qui donne le plus de contrôle sur la gestion du threading.

La deuxième méthode consiste à étendre la classe `CacheImpl` qui implémente l'interface `Cache` en y ajoutant les codes de bloquage.

```
public class ThreadSafeCacheImpl extends CacheImpl {

  synchronized public void put(Object key, Object value) {
    super.put(key, value);
  }

  synchronized public void put(Object key, Object value, long ttl) {
    super.put(key, value, ttl);
  }

  synchronized public Object get(Object key) {
    return super.get(key);
  }

  synchronized public Object remove(Object key) {
    return super.remove(key);
  }

  synchronized public void clear() {
    super.clear();
  }

  synchronized public void evict() {
    super.evict();
  }

}
```

Listing 3.9: Cache thread-safe par héritage

Dans ce cas, le code métier du cache n'ayant rien a voir avec la gestion du thread, la maintenance en est plus aisée. Par contre, il faudra étendre toutes les classes qui implémentent l'interface `Cache`, ce qui peut s'avérer fastidieux.

La troisième méthode emploie le pattern décorateur. Le pattern décorateur est « un motif de conception permettant d'attacher des fonctionnalités à des objets à l'exécution. Ce pattern se montre plus flexible que l'héritage, qui lui est statique »[Geary, 2001].

```
public class ThreadSafeCacheDecorator implements Cache {

  private Cache cache;

  public ThreadSafeCacheDecorator(Cache cache) {
    this.cache = cache;
  }

  synchronized public void put(Object key, Object value) {
    cache.put(key, value);
  }

  synchronized public void put(Object key, Object value, long ttl) {
    cache.put(key, value, ttl);
  }

  synchronized public Object get(Object key) {
```

```
    return cache.get(key);
  }

  synchronized public Object remove(Object key) {
    return cache.remove(key);
  }

  synchronized public void clear() {
    cache.clear();
  }

  synchronized public void evict() {
    cache.evict();
  }

}
```

Listing 3.10: Cache thread-safe par décorateur

Ce pattern permet de greffer la gestion des threads à n'importe quelle classe implémentant l'interface `Cache`. Il est utilisé de cette façon :

```
Cache cache = new ThreadSafeCacheDecorator(new CacheImpl());
```

Le choix d'une de ces techniques se fait essentiellement sur base du temps que l'on est prêt à y consacrer et sur le niveau de performance que l'on veut obtenir. En effet, plus on choisit une solution simple et générique, moins elle sera précise et efficace.

3.7 Cache distribué

Dans des environnements distribués comme des clusters, il est utile de mettre en commun de l'information ; c'est particulièrement vrai pour les fermes de serveurs webs. Les caches distribués y sont par exemple utilisés pour partager les sessions des utilisateurs.

On reprend ici l'idée du multi-thread à un niveau supérieur. Un tel cache est bien sûr plus difficile à réaliser car il fait intervenir des réseaux comme intermédiaires. Ceux-ci ajoutent des temps de latence importants et des possibilités d'erreurs supplémentaires.

L'une des difficultés majeures rencontrées par ce système vient de la cohérence des informations distribuées. En effet, si un objet caché vient à être modifié, la modification doit être répercutée à tous les processus qui l'utilisent. Il faut alors avoir recours à des procédés de bloquage complexes qui sortent du cadre de ce mémoire [JBoss, 2006].

CHAPITRE 4

SOLUTION CONCRÈTE

Lorsqu'un programmeur démarre un projet, il est confronté à un certain nombre de choix critiques. Il doit en effet déterminer les parties du développement qu'il effectuera effectivement et les parties qui s'appuieront sur des bibliothèques existantes.

Il faut savoir que la réalisation d'un système de caching est un processus assez complexe si l'on veut obtenir un résultat probant. Le choix se fera en fonction du type d'application que l'on veut programmer.

On peut distinguer deux grands types d'applications : d'une part les applications dont le caching occupe une place centrale comme un proxy sur le web et d'autre part les applications qui ne font qu'utiliser du caching comme outil et dont les objectifs sont tout autres comme la suite de Fibonacci.

Dans le premier cas, il sera sans doute utile de développer une solution particulière. Dans le second cas, le programmeur aura tout intérêt à s'appuyer sur une librairie déjà fonctionnelle.

4.1 Choix d'un standard

De nombreuses solutions de caching sont disponibles sur le marché, qu'elles soient opensources ou non [Java-Source.net, 2006a]. Chacune d'entre elles

dispose d'atouts propres et est parfois adaptée à un problème bien particulier.

La principale difficulté que l'on rencontre avec ce genre de solutions est qu'elles ne sont pas vraiment compatibles entre elles. Heureusement, il existe un début de spécification d'un standard pour le caching en Java. Ce standard est identifié par la référence JSR-107, aussi connu sous le nom moins barbare de jcache (Java Caching API)[briangoetz, 2005].

L'intérêt du choix d'un standard par rapport à une solution spécifique vient du fait qu'il permet de ne pas être lié à un fournisseur particulier. Changer de fournisseur revient à remplacer une bibliothèque par une autre. Le code de l'application qui l'utilise n'est pas modifié. On a donc alors la possibilité de choisir une implémentation en fonction des services offerts ou des performances que l'on veut obtenir.

Le JSR-107 est une API de caching basée sur l'interface `Map`[1]. Cette interface a l'avantage d'être bien connue des programmeurs et d'être très facile à utiliser.

Le diagramme de classe de jcache est la figure 4.1 à la page 25. Les différentes entités du diagramme correspondent à peu de choses près aux fonctionnalités présentées dans les précédents chapitres. Il ne s'agit bien sûr pas d'un hasard, j'ai préféré aborder les fonctionnalités du caching à l'aide de ce standard plutot que développer une API *ex nihilo*.

4.2 Contraintes et défauts du standard

Le choix du JSR-107 n'est pas sans problème. Celui-ci a en effet quelques imperfections qu'il faut connaître.

Par exemple, le langage Java comporte certains défauts de conception. L'un des plus ennuyeux est le problème de la copie d'objet. En effet, lorsque l'on passe un objet en paramètre à une méthode, cet objet n'y est pas copié mais référencé : une modification apportée à un objet venant d'un cache qui ne gère que des références, est en réalité apportée à l'objet stocké dans le cache. Il s'agit ici du problème bien connu de passage par valeur-référence[Eckel, 2006].

La constance d'un objet dans le cache n'est pas déterminée par le JSR-107 mais par ses implémentations. Une solution possible consiste en l'utilisation de la méthode `clone`[2] d'un objet. Malheureusement, cette méthode requiert

[1] `http://java.sun.com/j2se/1.5.0/docs/api/java/util/Map.html`
[2] `http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Object.html`

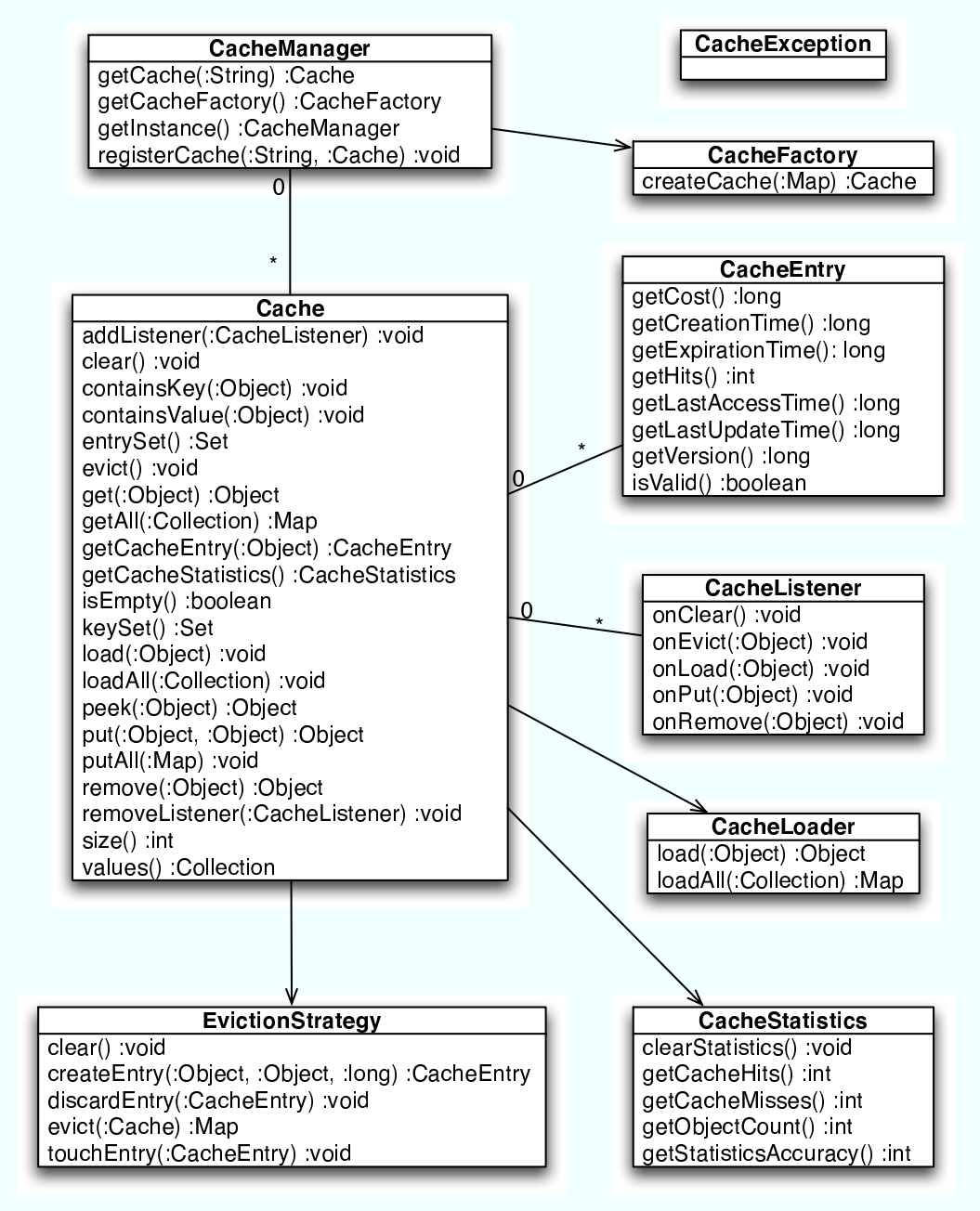

FIG. 4.1: Diagramme de classe du JSR-107 (jcache)

du code spécifique pour chaque type d'objets à cloner. Une autre implémentation possible consiste à conserver les objets sous forme de tableaux d'octets grâce à la sérialisation. Cette solution, si elle nécessite toutefois l'utilisation de l'interface `Serializable`[3], a l'avantage de pouvoir déterminer précisément l'espace mémoire utilisé par un objet caché, ce qui est impossible par référence ou par clonage.

Du point de vue de l'API, certaines méthodes ne sont pas claires ou ont des conséquences fâcheuses sur les performances de leurs implémentations. Prenons pour exemple la méthode `remove` de l'interface `Cache`, qui supprime un objet stocké dans le cache. La documentation disponible ne précise pas si cette méthode doit renvoyer l'ancienne valeur cachée, ce qui serait le comportement normal d'une `Map`. L'implémentation de cette fonctionnalité en mémoire ne pose aucun problème. Par contre, une implémentation par fichier serait grandement pénalisée. En effet, ce système serait obligé de lire systématiquement la donnée stockée dans un fichier avant de la supprimer et ce, même si l'on n'en a pas besoin.

De plus, certaines fonctionnalités sont incomplètes. C'est le cas du `CacheLoader` : contrairement à la méthodologie de récupération de données à partir d'un niveau inférieur, aucune indication n'est fournie quant à l'ajout de données dans ce niveau.

Par ailleurs, la structuration des données autour d'une `Map`, bien que simple, n'est pas forcément la plus pratique. Il existe d'autres systèmes basés sur des hiérarchies de clés qui offrent des possibilités intéressantes de traitement comme le parcours des clés ou la suppression en cascade de tous les fils d'un noeud [Smuts, 2005]. En généralisant davantage, on pourrait aussi imaginer un système basé sur la théorie des graphes.

D'autre part, le JSR-107 est une spécification encore à l'état de brouillon. Il est donc possible et même probable qu'elle soit modifiée ou complétée prochainement.

En sus, il apparaît aussi que les promoteurs de cette spécification ne sont actuellement pas très actifs. Les dernières spécifications disponibles datent en effet de janvier 2005.

Enfin, il n'existe pas d'implémentation sérieuse de cette spécification. La plupart des fournisseurs de solution de cache affirment que leur solution est similaire au standard. Le problème est qu'elles ne sont pas identiques et que donc les appels de fonctions sont spécifiques à chaque produit. Il ne sera alors

[3] `http://java.sun.com/j2se/1.5.0/docs/api/java/io/Serializable.html`

pas trivial de remplacer une solution de caching par une autre, ce qui aurait été le cas si elles avaient réellement implémenté la spécification.

Ces problèmes évoqués plus haut sont, pour la plupart, dus à une stagnation du développement du standard plutôt qu'à des défauts de conception.

4.3 Solution proposée

Il apparaît, au vu de ce qui a été développé dans les paragraphes précédents que le programmeur dispose de deux grands choix : soit utiliser une solution spécifique mais être alors lié à celle-ci, soit utiliser la spécification JSR-107.

Le deuxième choix, bien que préférable, ne sera pas exploitable sans un minimum de travail. Il faudra en effet que le programmeur redirige les méthodes du standard vers les méthodes spécifiques d'une solution de caching. Ce travail est heureusement bien moins important que d'implémenter complètement un système de cache à partir de zéro.

Quelque soit le choix du programmeur, les fonctionnalités présentées dans les précédents chapitres s'y retrouveront. Par contre, pour une question de simplicité, les prochains chapitres traitant de l'intégration du caching dans le processus de développement, s'appuieront sur le JSR-107 pour s'illustrer.

4.4 Problématique du temps réel

La programmation temps réel est un domaine à part en informatique. Elle concerne principalement les systèmes critiques qui requièrent une faible latence et les équipements embarqués qui disposent de ressources limitées, que ce soit en mémoire ou en traitements.

La progammation temps réel en Java est définie par la spécification RTSJ[4]. Celle-ci est articulée autours de la capacité à répondre à un événement du monde réel de manière fiable et déterministe. Il s'agit surtout de maîtriser le temps plutôt que de viser la rapidité.[java.sun.com, 2006]

Techniquement parlant, ces exigences peuvent être suivies en :

[4] https://rtsj.dev.java.net/

- diminuant le temps d'exécution dans le pire des cas[5]. Ces temps sont indéterminables et surviennent lors de la compilation just-in-time, du fonctionnement du ramasse-miettes[6] et de l'initialisation des classes.
- adaptant au problème du temps réel les bibliothèques de thread, d'ordonnancement et de synchronisation.
- utilisant des bibliothèques spécifiques permettant l'accès direct à la mémoire et la gestion asynchrone d'événements extérieurs.
- réduisant la création d'objets et le recours au ramasse-miettes, le coût d'allocation d'objets de taille importante étant significatif. Ceci peut être réalisé en recyclant de façon transparente des objets et/ou en les pré-allouant au démarrage.

Du point de vue du caching, certaines adaptations sont donc nécessaires.

Par exemple, l'implémentation la plus simple du JSR-107 repose sur l'extension le la classe `HashMap`[7] du JRE. Si cette classe convient parfaitement pour une utilisation courante, elle pose problème dans le domaine du temps réel. En effet, elle ne garantit pas un temps constant lors de la récupération d'un objet. Ce temps est variable et augmente en fonction de sa taille. De plus, chaque ajout d'une paire clé-objet crée un objet descripteur correspondant. Il n'y a pas de recyclage d'objet, ce qui entraîne une gestion de mémoire non optimale.

Malheureusement, une solution de caching temps réel basée sur le JSR-107 n'existe pas actuellement. Il est donc nécessaire de la programmer à l'aide de bibliothèques spécifiques comme par exemple Javolution[8] et d'appliquer certaines techniques pour réutiliser autant que possible les objets. Le tout doit tourner sur une machine virtuelle temps réel.

[5] worst-case execution time
[6] garbage collector
[7] http://java.sun.com/j2se/1.5.0/docs/api/java/util/HashMap.html
[8] http://javolution.org/

Deuxième partie

Intégration dans le processus de développement

CHAPITRE 5

SCÉNARIO D'INTÉGRATION

La première partie de cet ouvrage présente le caching et les différents services qu'il peut rendre. La deuxième partie décrit quant à elle l'intégration concrète du caching dans le processus de développement.

Cette intégration se fera autour de deux axes principaux : la localisation du code optimisable et les techniques concrètes d'intégration. Cette deuxième partie se terminera en développant une nouvelle technique d'intégration basée sur le caching dynamique de méthodes.

5.1 Illustration des concepts

Pour exemplifier les différents concepts des prochains chapitres, une petite application, appelée FTB, va se voir adjoindre un cache dans une tentative d'amélioration des performances.

L'application FTB est très simple et a pour but de classifier les fichiers d'un répertoire et de ses sous-répertoires en fonction de leurs catégories (audio, video, bureautique, ...) ou leurs extensions (.exe, .doc, .avi, ...).

L'idée de base de cette application est issue du problème courant de l'informatique qu'est le manque d'espace disque. En effet, bien que l'espace disque d'un ordinateur ne cesse d'augmenter grâce à la diminution du coût de stockage et à l'amélioration des techniques, celui-ci est vite saturé par le téléchar-

gement de ressources sur Internet ou encore le montage video. L'utilisateur a donc besoin tôt au tard de faire le tri dans ses fichiers pour libérer de la place.

Les programmes traditionnels fournis avec un système d'exploitation analysent les fichiers sur base de leur localisation dans une hiérachie de répertoires, ce qui permet de déterminer l'espace utilisé par tous les fichiers d'un même répertoire. Cette méthode est malheureusement assez limitée.

La classification réalisée par FTB se porte quant à elle sur les catégories et les extensions, indépendamment de leur localisation. Il devient ainsi possible de connaître le poids d'une catégorie par rapport à une autre, sur base du nombre total de fichiers qui la composent ou sur base de la taille totale de ces mêmes fichiers.

Le tri nécessaire à cette classification s'effectue de la façon suivante : l'application parcourt le contenu d'un répertoire sélectionné par l'utilisateur. Pour chaque fichier rencontré, elle en extrait l'extension et recherche le type correspondant dans un fichier xml. Chaque fichier ainsi identifié est ajouté dans une hiérarchie organisée selon les critères de catégories et d'extensions. Celle-ci a pour modèle la figure 5.1 de la page 32 et sert à générer des rapports sous forme de tableaux. Le fonctionnement de FTB est représenté par la figure 5.2 à la page 33.

Les différents écrans que l'on veut obtenir à partir de cette classification sont les tableaux affichant :

- la liste des extensions avec pour chacune d'entre elles, le nombre de fichiers qu'elle contient et la somme de leur taille.
- la liste des catégories, le nombre de fichiers qu'elles contiennent et la somme des tailles de ces fichiers.

Un clic sur une extension ou une catégorie affichera aussi la liste des fichiers lui appartenant.

Une étude rapide de l'architecture de cette application met en évidence deux fonctionalités qui pourraient bénificier d'un caching. Il y a d'une part la somme des tailles des fichiers selon les différents critères demandés et d'autre part la récupération de la catégorie associée à une extension.

Dans le premier cas, chaque rafraîchissement d'un des tableaux obligerait l'application à recalculer cette somme. Sachant qu'un répertoire personnel peut facilement contenir des dizaines de milliers de fichiers, ce mode de fonctionnement amènerait une latence importante dans l'affichage des résultats.

Dans le deuxième cas, on constate qu'en général la plupart des fichiers d'un même répertoire sont d'un type commun. En retenant dans un cache les

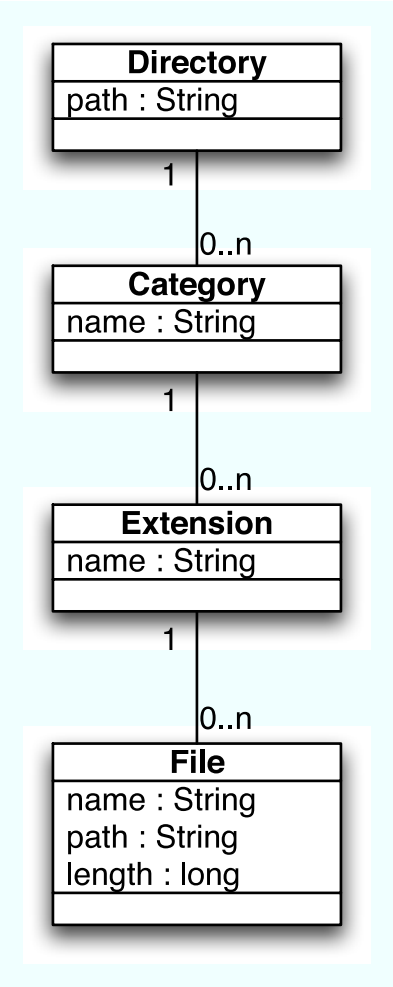

FIG. 5.1: Diagramme de classes de l'application FTB

types les plus courants, on pourrait grandement réduire les accès au fichier xml, qui sont communément considérés comme lents.

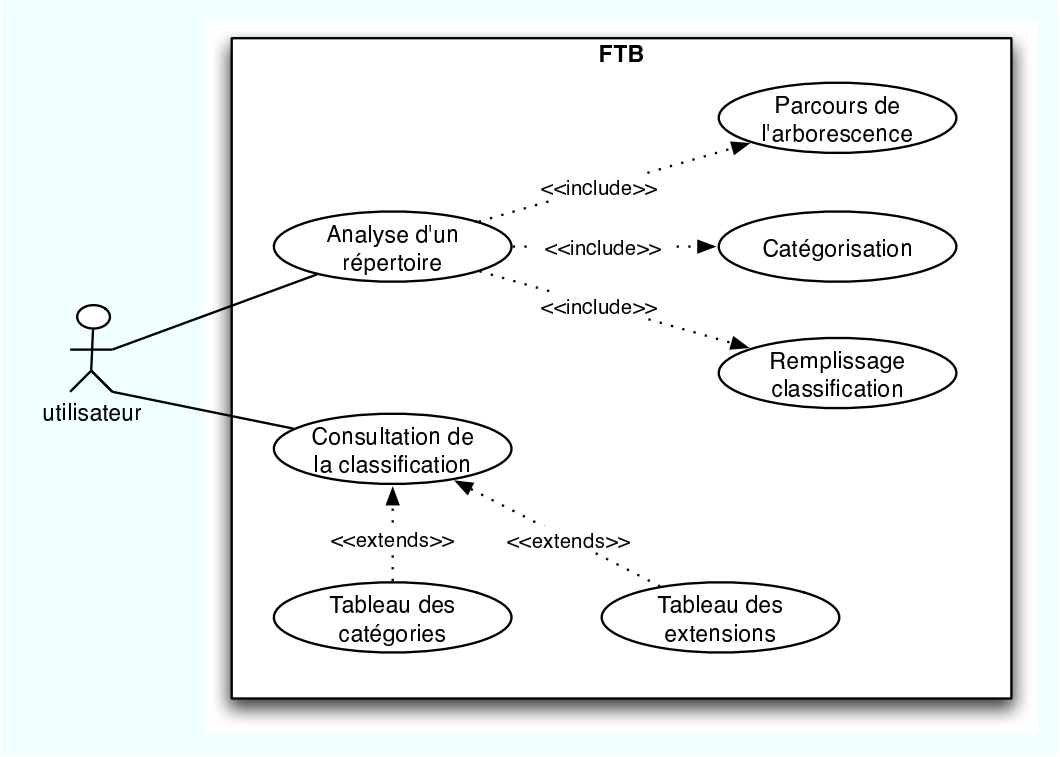

FIG. 5.2: Diagramme des cas d'utilisation de l'application FTB

CHAPITRE 6

LOCALISATION DU CODE OPTIMISABLE

Lorsque l'on veut optimiser du code à l'aide de caching, il faut au préalable localiser le code susceptible d'être amélioré. Il n'est pas toujours indiqué d'apporter une telle modification et le but de ce chapitre est justement de définir certaines règles permettant l'identification des portions de code à même d'en tirer profit.

6.1 Remarques sur l'optimisation

Lors de la réalisation d'une application, il est important de prendre en compte les éventuelles optimisations de performances et de les appliquer aux moments opportuns du cycle de développement. Omettre ces améliorations peut s'avérer très coûteux par la suite. En effet, un problème créé ou non corrigé dans une phase du cycle de développement requiert beaucoup plus d'efforts pour être modifié dans les phases suivantes, chaque altération en entraînant une autre à la façon d'une boule de neige.

Par ailleurs, le réglage des performances est une affaire de choix. En général, les différents éléments d'une application sont liés entre eux. La modification d'une partie affecte le comportement des autres et malheureusement pas forcément pour le meilleur.

L'optimisation durant la phase de design et d'analyse diffère de la phase d'implémentation. En général, un problème de performances engendré par un mauvais design est la source d'un important travail de correction. Par contre, du code produisant de faibles performances mais correctement conçu est plus simple à rectifier.

Durant l'étape d'implémentation, la bonne pratique consiste à se focaliser sur les fonctionnalités et sur la production de code exempt de bogues. Il est habituellement plus judicieux d'ignorer les performances lors de l'écriture de code. Le réglage des performances ne devrait être réalisé que lorque le code fonctionne correctement.[Shirazi, 2000]

En outre, beaucoup d'améliorations rendent le code source plus compliqué et plus difficile à lire. L'équilibre entre code performant, simple et lisible est délicat.

Toutefois, dans la plupart des programmes, 80% du temps passé à l'exécution est dépensé dans moins de 20% du code, et seulement 5% du code mérite d'être optimisé. Ces 5% représentent les goulots d'étranglement d'une application.[Shirazi, 2000]

La meilleure approche pour améliorer les performances consiste à se concentrer sur ces 5%. De cette manière, on commence par construire un programme le plus clairement possible sans se soucier des problèmes des performances. Les performances sont mesurées en fin de développement à l'aide d'un outil de profilage qui va identifier les zones sensibles qui méritent l'attention du programmeur. Il ne reste plus à celui-ci qu'à améliorer ces portions de code et ainsi limiter l'impact de ces modifications sur la lisibité et la complexité du code.[Fowler, 2004]

6.2 Critères généraux d'identification de code optimisable

Un code typiquement optimisable est un code appelé fréquemment. En effet, il serait inutile, voire même coûteux en performance, de mettre en place un système de cache pour du code qui serait utilisé une fois par an.

De plus, la réponse renvoyée par ce code ne doit pas varier ou alors doit être liée aux paramètres d'entrée. En effet, s'il peut être opportun de mettre en cache le résultat d'une fonction mathématique, il est ridicule de sauver la valeur de retour d'une méthode aléatoire.

Il est également important que les coûts de traitement induits par le système de caching ne dépassent pas ceux de la méthode originelle. Comme exemple, on peut citer des requêtes à des bases de données ou l'utilisation de webservices, ces derniers faisant intervenir à la fois des traitements extérieurs et de la latence réseau.

Le code doit en outre impérativement produire un résultat. Dans le cas contraire, il n'y aurait rien à mettre en cache.

De même, un code qui modifie des données quelque part ne peut pas être caché car ces données ne seraient plus jamais modifiées lors d'appels ultérieurs. Par exemple, si l'on reprend l'idée de mettre en cache le résultat de requêtes à des bases de données SQL, on s'aperçoit vite que seuls les requêtes de sélection peuvent être mis en cache tandis que les insertions et autres mises à jour n'ont aucun sens ici.

Par ailleurs, l'utilisation de caching peut introduire des problèmes au niveau de la sécurité et de l'audit. Un cache peut en effet intercepter une requète à une méthode et renvoyer la réponse sans exécuter les contrôles de sécurités éventuellement codés dans cette méthode.

Enfin, il est préférable d'éviter de mettre en cache des données temps réel tel que des valeurs de marchés financier et des données sensibles comme des mots de passe ou des numéros de sécurité sociale.[Ruiz, 2006]

6.3 Remarques concernant l'implémentation d'un cache

Un système de cache peut être réalisé de différentes façons. Cette implémentation détermine ses fonctionnalités et ses performances. En considérant que l'on implémente les mêmes fonctionnalités dans différents systèmes de cache, on se rend compte tout de même qu'une implémentation basée sur la mémoire vive d'un ordinateur sera beaucoup plus rapide qu'une implémentation basée sur un système de fichiers, mais disposera de beaucoup moins d'espace.

On peut représenter la performance vis-à-vis de l'espace disponible par une hiérarchie pyramidale (fig. 6.1, p. 37). Au sommet, on trouve les mémoires vives et, à la base, les mémoires de masse.

Le choix d'une implémentation se base sur deux critères : la durée de vie d'une donnée dans un cache et le temps de traitement requis pour la création de la donnée.

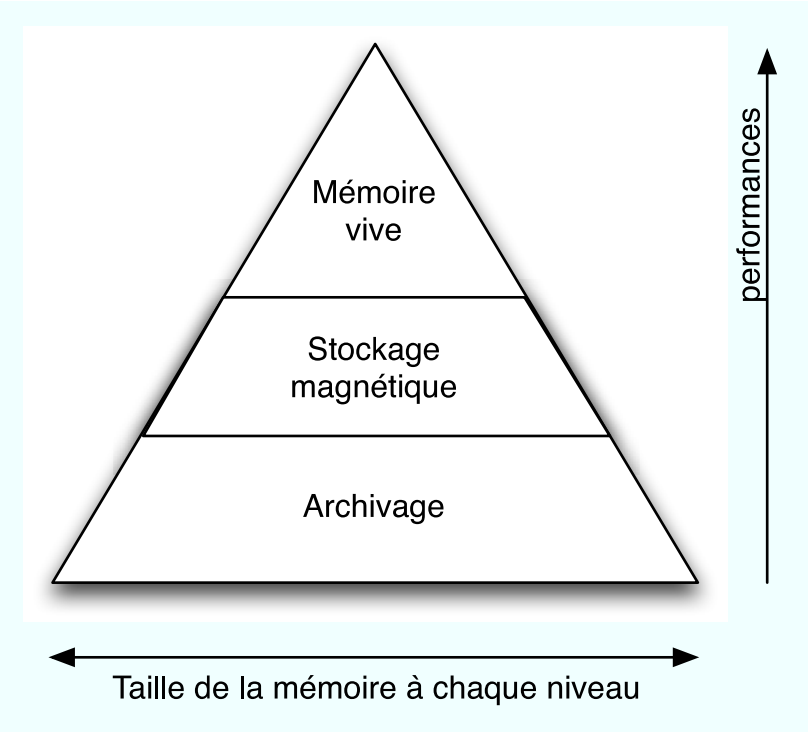

FIG. 6.1: Hiérarchie mémoire d'un ordinateur

Plus une donnée est volatile, plus on utilisera un système de faible capacité. De même, plus une donnée est longue à générer, plus on pourra se permettre d'utiliser un système de faibles performances.

6.4 Contre-performances liées à l'implémentation du cache

L'intérêt de l'utilisation d'un système de cache réside dans le gain de performances qu'il apporte à une application. Or, comme introduit dans les chapitres précédents, ces systèmes ont un coût en terme de traitements. Il faut donc choisir soigneusement le code que l'on va optimiser de la sorte.

Dans l'exemple FTB, on peut constater que les deux fonctionnalités optimisables répondent aux critères généraux définis plus haut. Plus particulièrement, la deuxième fonctionnalité liée aux accès à un fichier xml profite pleinement d'un système de caching car sa durée de traitement, c'est-à-dire de

recherche et de lecture des données, est grandement supérieure à la récupération dans un cache implémenté en mémoire.

Par contre, ce n'est plus vrai lorsque l'on utilise un système de cache basé sur un système de fichiers. En effet, la recherche et la lecture d'informations dans le fichier xml sera équivalente en temps à la recherche des données cachées dans un système de fichiers.

6.5 Coût de l'intégration

L'intégration d'un système de cache ne se fait pas sans un minimum de travail. Le plus gros du travail se situe dans la recherche du code optimisable et dans la calibration des paramètres d'optimisation.

Toutefois, certaines optimisations autres qu'un système de cache sont parfois plus efficaces car elles sont bien connues et robustes. Par exemple, dans le cadre du design d'une base de données, il est facile d'améliorer grandement les performances en y ajoutant une redondance contrôlée.

Ces techniques ne sont pas forcément plus performantes ou souples que le caching mais sont par contre mises en oeuvre bien plus facilement. Un programmeur peut avoir intérêt à troquer un peu en performance pour du gain en temps de développement.

La première fonctionnalité optimisable de FTB rentre parfaitement dans ce cadre. En effet, il est très simple de retenir la somme des tailles de fichiers à chaque niveau du diagramme de classe en y insérant un compteur redondant. Ces compteurs sont mis à jour lorsqu'un fichier est ajouté dans la structure. Cette solution s'avère plus performante que n'importe quel système de cache, même minimaliste.

6.6 Cas du modèle client-serveur

Le cas du modèle client-serveur mériterait à lui seul plusieurs chapitres de ce mémoire. Ce n'est d'ailleurs pas un hasard si une bonne part des systèmes de cache disponibles lui sont consacrés et sont conçus exclusivement dans cette optique.

Son application la plus connue est le web caching. Le web caching consiste à « stocker de manière temporaire des documents web (pages HTML, images) pour réduire l'usage de la bande passante, diminuer la latence et alléger la charge des serveurs »[Davison, 2006].

Dans ce modèle (fig. 6.2, p. 39), le ou les caches peuvent être localisés à plusieurs endroits à la fois, que ce soit du coté client (le cache d'un navigateur), du coté serveur (un module de cache intégré à Apache par exemple) ou à l'intérieur même du réseau à l'aide d'intermédiaires (les fameux proxies).

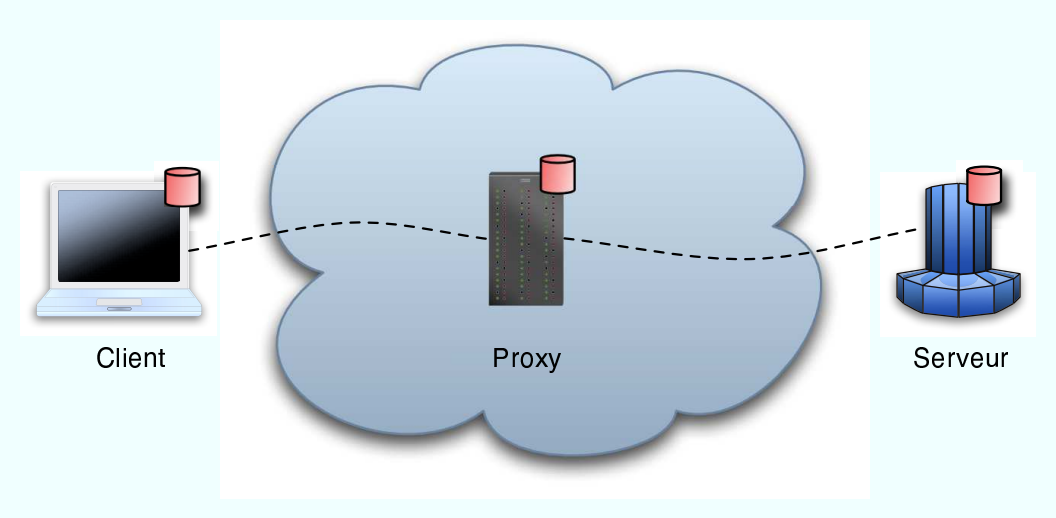

FIG. 6.2: Web caching

Revenons à un modèle plus général de client-serveur (fig. 6.3, p. 40). On peut représenter de manière abstraite le cache comme un intermédiaire transparent. Chaque appel du client est relayé au serveur par le ou les caches si la donnée n'y est pas disponible.

La mise en place d'un cache du coté serveur et/ou client permet d'économiser du temps de traitement. Si le cache du coté client a en outre l'avantage d'éviter les problèmes de latence réseau, le cache du coté serveur permet de le centraliser et ainsi optimiser globalement l'espace mémoire requis.

La centralisation ou non du cache a aussi une conséquence non négligeable sur la gestion de la cohérence des données cachées. En effet, si une donnée est mise à jour, il est très facile d'invalider le cache du coté serveur puisqu'il gère lui-même cette mise à jour. Par contre, le client, s'il n'est pas à l'origine de cette mise à jour, n'est pas au courant de celle-ci. Il pourrait en résulter l'affichage de données périmées avec toutes les conséquences qui peuvent en découler.

Le cas du web caching est ici intéressant car il intégre la gestion de l'expiration du cache directement dans son propre protocole. Il suffit de consulter les entêtes du protocole http pour s'en rendre compte.

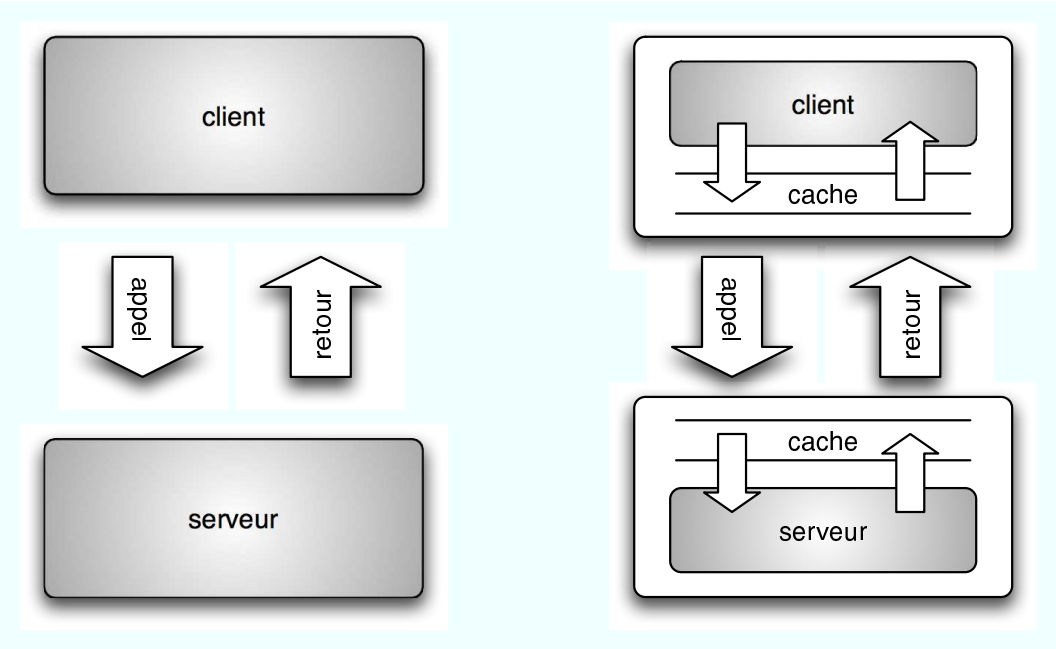

FIG. 6.3: Modèle client-serveur avec et sans cache

En géréral, il faut donc que le programmeur pense à gérer d'une façon ou d'une autre les expirations de cache du coté client. Une solution est de contourner le problème en ne cachant que les données stables d'un application car elles ne varient pas ou peu.

Les éléments développés ici partent du principe que les données d'un cache doivent toujours être cohérentes avec les données réelles. Ce n'est toutefois pas toujours le cas. Dans le cadre d'un moteur de forum sur le web par exemple, il n'est pas nécessaire de montrer la dernière version d'un sujet. La plupart des moteurs de forum mettent en cache pendant quelques secondes les différentes pages qu'ils produisent, ce qui permet déjà d'augmenter significativement la disponibilité du serveur. Cette incohérence n'est pas gênante car l'utilisateur ne s'en rend généralement pas compte.

CHAPITRE 7

TECHNIQUES CONCRÈTES D'INTÉGRATION

L'exemple de la suite de Fibonacci introduit au premier chapitre, montre que l'ajout du système de cache passe par une certaine modification du code source. Cette modification peut être plus ou moins importante selon le contexte et la méthode employée.

La conséquence malheureuse de ce changement est que le code métier est alors altéré par du code qui ne lui est pas lié, ce qui en diminue la lisibité et complique la maintenance. Habituellement, plus un code est optimisé, plus il s'éloigne de l'algorithme général.

Heureusement, il est possible de limiter ce problème en employant certaines techniques d'intégration décrites ci-dessous. Les sections suivantes vont en expliquer les mécanismes et en extraire les points forts et faibles ainsi que les domaines d'application respectifs.

7.1 Modification brute du code

La technique la plus simple consiste à modifier directement le code source à l'endroit où l'on a besoin de la donnée cachée.

Bien que n'employant pas de véritable pattern, il est tout de même possible de dégager une structure récurrente dont le diagramme de processus est la figure 7.1 à la page 43. Il s'agit donc de vérifier si une donnée est disponible dans le cache. Si oui, elle est récupérée, sinon, le programme la crée, la stocke dans le cache et enfin renvoie le résultat.

En appliquant cette technique dans l'exemple de la suite de Fibonacci, on obtient :

```
public class Fibonacci {

  private static final Cache cache = CacheManager.getInstance()
      .getCache("fibo");

  public int fibonacci(int n) {
    // Demande
    Integer result;
    // Existe dans le cache
    if (cache.containsKey(n)) {
      // Recuperation du cache
      result = (Integer) cache.get(n);
    } else {
      // Creation
      if (n < 2)
        result = n;
      else
        result = (fibonacci(n - 1) + fibonacci(n - 2));
      // Sauvegarde dans le cache
      cache.put(n, result);
    }
    // Envoi
    return result;
  }
}
```

Listing 7.1: Fibonacci optimisé par modification brute (v1)

Cette structure, bien que très logique, n'est pas utilisée habituellement en Java. En effet, ce langage dispose d'une astuce qui permet d'alléger le code et d'accélérer légèrement le traitement. Cela se fait grâce à l'utilisation du mot clé `null`. Au lieu de vérifier si une donnée est disponible dans le cache, on la récupère directement. On vérifie ensuite si cette donnée est `null` ou pas. Si oui, il faut la recalculer et la stocker dans le cache. Le diagramme de processus correspondant est la figure 7.2 à la page 44.

Le code de l'exemple de Fibonacci est modifié comme suit :

```
public class Fibonacci {

  private static final Cache cache = CacheManager.getInstance()
      .getCache("fibo");

  public int fibonacci(int n) {
    // Demande
```

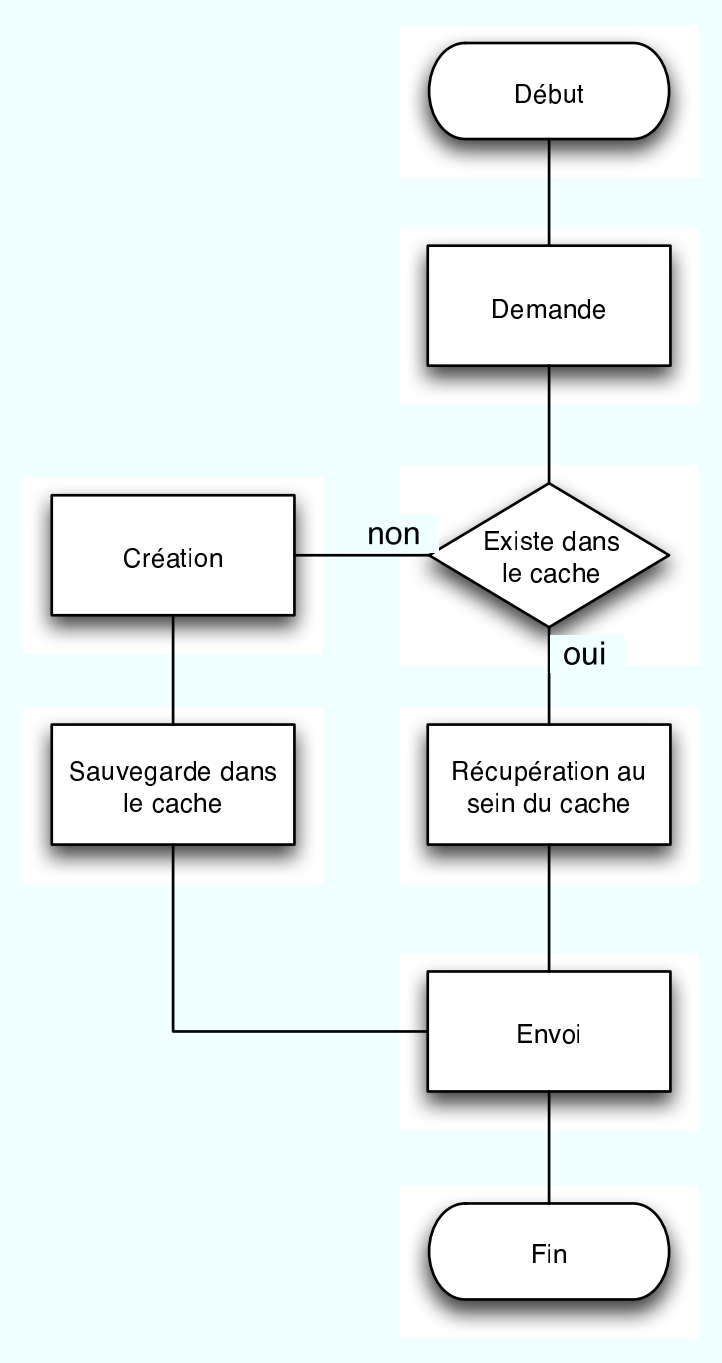

FIG. 7.1: Diagramme de processus de l'intégration par modification brute (v1)

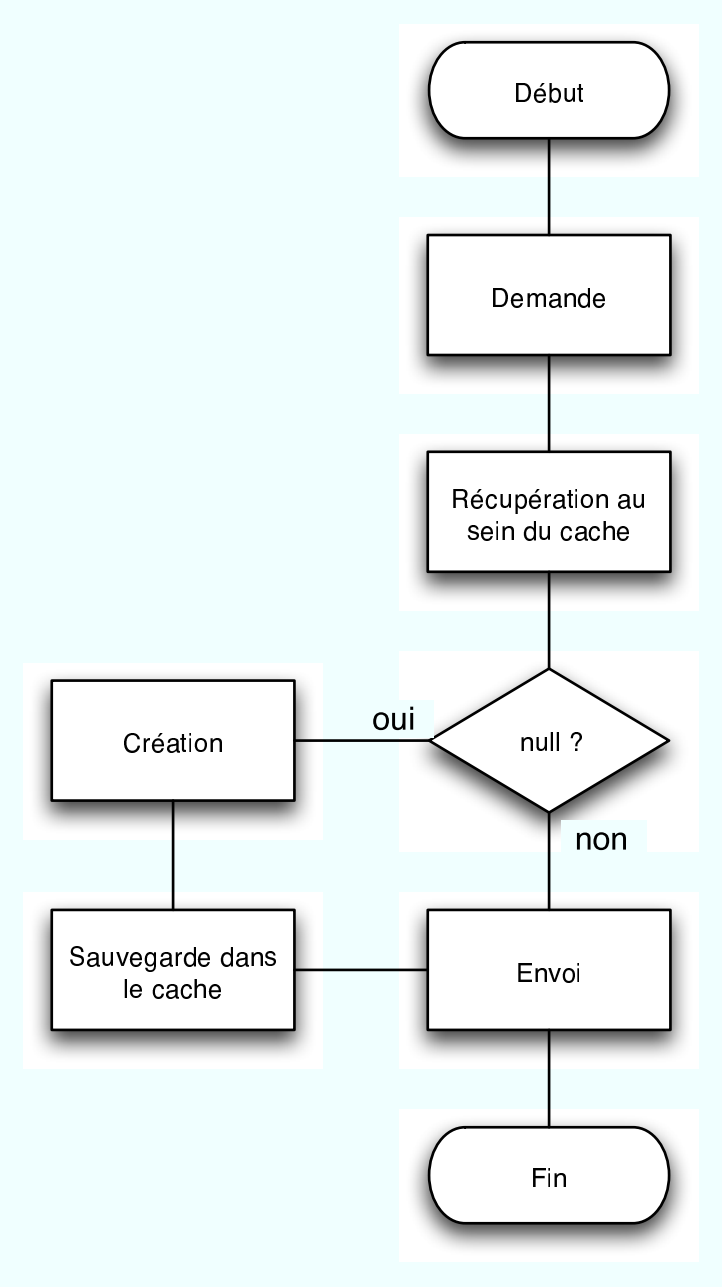

FIG. 7.2: Diagramme de processus de l'intégration par modification brute (v2)

```
    Integer result = (Integer) cache.get(n);
    // Recuperation du cache
    if (result == null) {
      // Creation
      if (n < 2)
        result = n;
      else
        result = (fibonacci(n - 1) + fibonacci(n - 2));
      // Sauvegarde dans le cache
      cache.put(n, result);
    }
    // Envoi
    return result;
  }
}
```

Listing 7.2: Fibonacci optimisé par modification brute (v2)

Cette optimisation est efficace mais présente un petit défaut : il n'est pas possible de stocker dans le cache la valeur `null`.

Cette technique est donc fort simple à mettre en oeuvre. N'importe quel programmeur peut choisir de l'utiliser selon ses besoins lors de l'implémentation du code de son application.

Par contre, elle a le gros inconvénient de modifier passablement le code qu'elle veut optimiser. Il y a donc un risque que le code métier soit noyé sous le code d'appel au cache. On perd alors de la lisibilité.

D'autre part, cette technique n'est en rien publique dans le sens ou il n'est pas possible de détecter sa présence sans une documentation précise de la part du développeur. Autrement dit, il arrive souvent que l'on dédouble un système de cache simplement parce qu'on n'est pas au courant de l'existence d'un cache dans une bibliothèque que l'on emploie. Non seulement, on réalise deux fois le travail, mais en plus il y a de fortes chances que l'on pénalise l'application par l'ajout de traitements inutiles.

7.2 Pattern de caching

Pour pallier le problème de la publicité d'un caching introduit à la section précédente, il est possible d'avoir recours à un pattern spécialisé.

Il existe plusieurs patterns de caching. Nous nous intéresserons ici au pattern de « cache management »[Grand, 2003].

Ce pattern consiste en la délégation de la récupération d'un objet par une classe `Client` à une classe dédiée appelée `CacheManager` sur base d'un identifiant nommé `ObjectKey`. La classe `CacheManager` commence par demander

l'objet à la classe `Cache`. Si l'objet est présent, il est renvoyé à la classe `Client`, sinon, `CacheManager` délègue la création de l'object à la classe `ObjectCreater`, ajoute l'object nouvellement créé dans le cache et le renvoie à la classe `Client`. (fig. 7.3, p. 46)

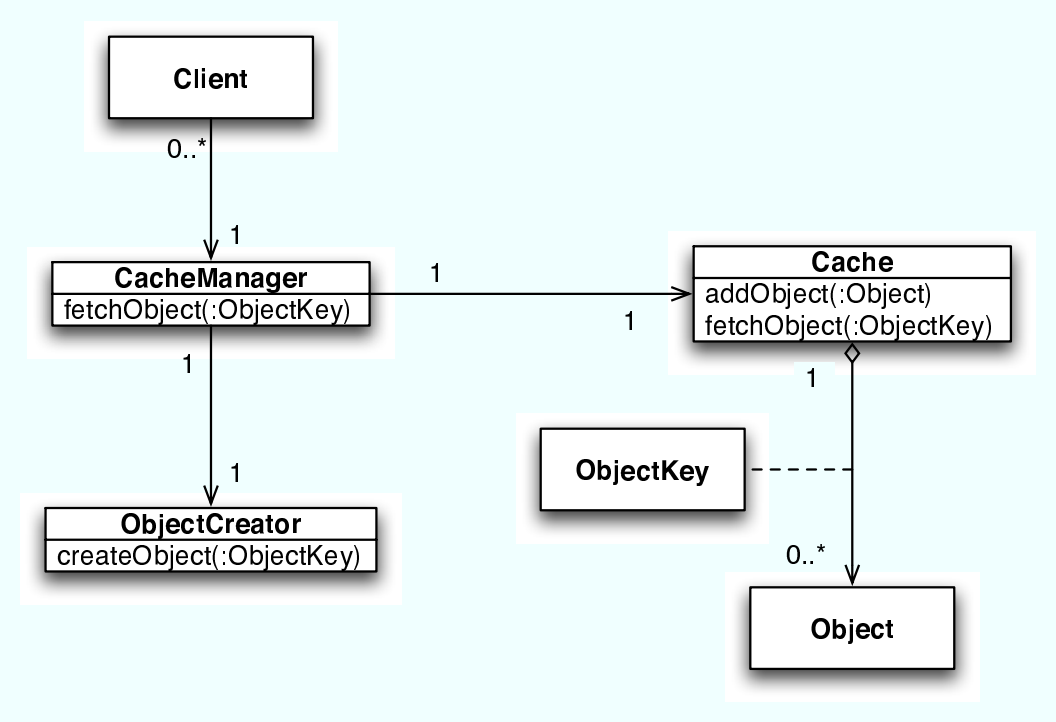

FIG. 7.3: Diagramme de classe du pattern cache management

Les rôles des différentes classes sont donc clairement définis dans ce pattern. Le résultat est alors presque auto-documenté par sa structure et est simple à programmer, chaque classe n'ayant que des fonctions limitées. La classe `Cache` peut d'ailleurs être reprise d'une solution de caching externe, ce qui diminue le code à produire.

Tel quel, cette technique présente l'inconvénient de devoir être planifiée au niveau de l'analyse de l'application et non pas à l'implémentation.

De plus, elle ajoute un certain nombre de classes inutiles du point de vue métier. Ces classes peuvent prendre alors beaucoup de place et ainsi de nouveau noyer le code important dans la masse.

Ces problèmes pourraient être réduits en faisant en sorte que la classe `CacheManager` étende la classe `ObjectCreater`. Malheureusement, on se heurte alors aux problèmes d'héritage multiple de classes en Java.

Cette solution n'est donc pas très flexible.

7.3 Décorateur classique

Le pattern décorateur, introduit au troisième chapitre, permet ici d'augmenter la flexibilité de la solution de pattern. Il s'agit simplement de séparer le code métier du code d'appel au cache tout en masquant son existence.

Cela est effectué en quelques étapes. Il faut d'abord définir une interface reprenant l'ensemble des méthodes publiques de la classe que l'on veut intégrer au cache. Ensuite, il faut créer deux classes qui implémentent cette interface. La première contient le code métier. La seconde est un décorateur qui maintient un lien vers la première classe et qui gère les appels au cache.

Le code résultant, appliqué à l'exemple de Fibonacci, est le suivant :

```
public interface Fibonacci {
  public int fibonacci(int n);
}
```

Listing 7.3: Interface de Fibonacci

```
public class FibonacciImpl implements Fibonacci {

  public int fibonacci(int n) {
    if (n < 2)
      return n;
    return (fibonacci(n - 1) + fibonacci(n - 2));
  }

}
```

Listing 7.4: Implémentation de Fibonacci

```
public class FibonacciCacheDecorator implements Fibonacci {

  private Fibonacci delegate;

  private Cache cache;

  public FibonacciCacheDecorator(Fibonacci delegate) {
    this.delegate = delegate;
    cache = CacheManager.getInstance().getCache("fibo");
  }

  public int fibonacci(int n) {
    Integer result = (Integer) cache.get(n);
    if (result == null) {
      result = delegate.fibonacci(n);
      cache.put(n, result);
    }
    return result;
  }
```

```
}
```

Listing 7.5: Décorateur classique ajoutant du caching à Fibonacci

Enfin, l'appel à la méthode de la classe Fibonacci se fait simplement de la façon suivante :

```
Fibonacci fibo = new FibonacciCacheDecorator(
    new FibonacciImpl());
System.out.println(fibo.fibonacci(3));
```

Listing 7.6: Exemple d'appel au décorateur classique

L'utilisation du pattern décorateur dans ce cas amène toutefois quelques problèmes. D'une part, il requiert l'existence d'une interface à implémenter, d'autre part, il est non réutilisable. Il faut réécrire une classe spécialisée pour chaque classe que l'on désire cacher.

Un troisième problème vient aussi ternir le tableau. Il s'agit de la multiplication excessive des classes du programme. En effet, au lieu d'avoir une classe métier, on a maintenant trois classes distinctes.

Heureusement, la flexibilité du pattern décorateur permet de compenser ces problèmes. Il est très facile d'ajouter ou de retirer un cache d'une application. Cela peut donc être réalisé à n'importe quelle étape du cycle de développement.

7.4 Décorateur générique

Le problème de la non réutilisabilité de la solution précédente peut être résolu par l'emploi d'une fonctionnalité propre à Java. Il s'agit des classes "Dynamic proxy". Contrairement aux classes Java normales où les interfaces implémentées sont connues et fixées à la compilation, il est possible d'écrire une classe générique qui déterminera à l'exécution les interfaces qu'elle implémente [White, 2003].

Une classe dynamic proxy est créée en deux étapes. La première consiste à créer une classe qui implémente l'interface `InvocationHandler`[1] et qui redirige les appels de méthodes vers la classe qui s'occupe du traitement. La seconde n'est que l'instanciation du proxy à l'aide de la méthode statique `Proxy.newProxyInstance`[2].

[1] http://java.sun.com/j2se/1.5.0/docs/api/java/lang/reflect/InvocationHandler.html

[2] http://java.sun.com/j2se/1.5.0/docs/api/java/lang/reflect/Proxy.html

```
public class SimpleHandler implements InvocationHandler {

  protected final Object delegate;

  public SimpleHandler(Object delegate) {
    this.delegate = delegate;
  }

  /**
   * Les appels de methodes sont rediriges vers une autre classe
   * (delegate) qui s'occupe du traitement
   */
  public Object invoke(Object proxy, Method method, Object[] args)
      throws Throwable {
    try {
      return method.invoke(delegate, args);
    } catch (InvocationTargetException e) {
      throw e.getTargetException();
    }
  }

}
```

Listing 7.7: Classe basique gérant les appels aux méthodes pour un proxy

```
    // instanciation de la classe qui s'occupe du traitement
    Fibonacci fiboImpl = new FibonacciImpl();

    // creation d'une instance du proxy
    Fibonacci fiboProxy = (Fibonacci) Proxy.newProxyInstance(
        fiboImpl.getClass().getClassLoader(),
        fiboImpl.getClass().getInterfaces(),
        new SimpleHandler(fiboImpl));

    System.out.println(fiboProxy.fibonacci(10));
```

Listing 7.8: Exemple d'instanciation d'un proxy basique

La classe `SimpleHandler` ne dispose pas encore de gestion de cache. Pour ce faire, il suffit d'ajouter le code habituel présenté dans les précédentes sections ainsi qu'un mécanisme s'assurant que la méthode à traiter renvoie bien un résultat (`isVoidMethod`).

D'autre part, vu que l'on met en cache le résultat de toutes les méthodes de l'interface, la clé d'indentification des données cachées est constituée de la méthode appelée et de la valeur de ses arguments (`buildKey`).

```
public class CacheHandler extends SimpleHandler {

  private final Cache cache;

  public CacheHandler(Object delegate) {
    super(delegate);
    cache = CacheManager.getInstance().getCache("generic");
  }
```

```
  @Override
  public Object invoke(Object proxy, Method method, Object[] args)
      throws Throwable {

    // inutile d'utiliser le cache si la methode ne renvoie rien
    if (Util.isVoidMethod(method))
      return super.invoke(proxy, method, args);

    // code de gestion du cache
    Object key = Util.buildKey(method.getName(), Arrays
        .toString(args));
    Object value = cache.get(key);
    if (value == null) {
      value = super.invoke(proxy, method, args);
      cache.put(key, value);
    }

    return value;
  }

}
```

Listing 7.9: Classe gérant les appels aux méthodes pour un proxy avec caching

L'utilisation de la nouvelle classe `CacheHandler` est identique à la précédente :

```
    // instanciation de la classe qui s'occupe du traitement
    Fibonacci fiboImpl = new FibonacciImpl();

    // creation d'une instance du proxy
    Fibonacci fiboProxy = (Fibonacci) Proxy.newProxyInstance(
        fiboImpl.getClass().getClassLoader(),
        fiboImpl.getClass().getInterfaces(),
        new CacheHandler(fiboImpl));

    System.out.println(fiboProxy.fibonacci(10));
```

Listing 7.10: Exemple d'intanciation d'un proxy avec caching

Enfin, pour simplifier le code à produire, la classe `GenericCacheDecorator` se charge de créer un proxy à l'aide de la méthode statique `decorate`. Le recours aux types génériques du langage Java permet d'éviter des transtypages ultérieurs inutiles.

```
public class GenericCacheDecorator {

  public static <X> X decorate(X object) {
    return (X) Proxy.newProxyInstance(object.getClass()
        .getClassLoader(), object.getClass().getInterfaces(),
        new CacheHandler(object));
  }

}
```

Listing 7.11: Décorateur générique ajoutant du caching

Le code résultant est ainsi bien simplifié et se résume à :

```
Fibonacci fibo = GenericCacheDecorator
    .decorate(new FibonacciImpl());

System.out.println(fibo.fibonacci(10));
```

Listing 7.12: Exemple d'utilisation du décorateur générique

On constate que l'appel à la méthode de la classe Fibonacci est similaire à la version de la section précédente. La seule différence réside dans l'utilisation d'une classe unique à la place d'une classe spécialisée.

Cette solution à base de décorateur générique n'est pas exempte de défauts. En effet, elle repose sur le système de réflexion de Java qui, même s'il s'est grandement amélioré dans les dernières versions, est moins rapide que des appels directs aux méthodes. Les performances sont donc moindres. En outre, il n'est pas possible de sélectionner les méthodes que l'on veut cacher : toutes, hormis celles qui ne renvoient rien, sont automatiquement cachées, ce qui peut provoquer des erreurs lors de mises à jour de données.

Enfin, la publicité du caching est faible. Il faut que le programmeur ait accès aux sources pour constater son utilisation.

CHAPITRE 8

CACHING DYNAMIQUE DE MÉTHODES

Les techniques décrites dans les précédents chapitres ont mis en évidence un certain nombre de problèmes récurrents dans l'intégration : la lisibilité du code métier, la publicité du caching, la planification fastidieuse, la multiplication excessive des classes et l'automatisation difficile.

Le but de ce chapitre est de mettre au point une nouvelle technique à l'aide des outils modernes qu'offre actuellement Java pour passer outre, ou tout du moins réduire sensiblement, ces problèmes.

8.1 Limitation du champ d'application

S'il est possible de cacher n'importe quoi, c'est-à-dire aussi bien des classes, des méthodes ou des extraits de code, il est plus judicieux de se limiter aux résultats de méthodes. En effet, une méthode représente souvent un bloc logique qui effectue une action, produit un résultat, voire les deux.

Cette idée est implicitement suggérée dans le précédent chapitre lors de l'utilisation du pattern décorateur. Pour fonctionner, ce pattern a besoin des interfaces des classes qu'il doit cacher. Or, ces interfaces décrivent les méthodes publiques de ces classes.

Par ailleurs, cette limitation a pour conséquence de ne pas être obligé d'altérer le code métier pour y ajouter du caching. On se contente simplement

d'enrober l'appel à la méthode avec une gestion de cache. Ceci peut être réalisé de manière plus ou moins transparente grâce au pattern décorateur.

Cette limitation du champ d'application n'est donc pas une contrainte mais plutôt une simplification du problème. Le code obtenu est beaucoup plus lisible et sa maintenance se voit ainsi simplifiée.

8.2 Méta-programmation par annotation

Lorque l'on ajoute du caching à des méthodes, on s'aperçoit vite que le code de manipulation de caching est presque toujours le même quelque soit le contexte. Seuls, le nom de la méthode non cachée et certains paramètres ou variables changent. Cette propriété est d'ailleurs exploitée dans la technique du décorateur générique introduit au précédent chapitre.

Malheureusement, cette technique comporte un défaut de taille : il n'est pas possible de choisir les méthodes qui doivent être cachées autrement que par l'interface que l'on a défini.

Pour résoudre cette faiblesse, il faudrait pouvoir désigner les méthodes dont on veut cacher le résultat. Une solution possible serait de passer la liste des méthodes à cacher en paramètre au constructeur du décorateur générique comme suit :

```
Fibonacci fibo = GenericCacheDecorator.decorate(
    new FibonacciImpl(), new String[] { "fibonacci" });
System.out.println(fibo.fibonacci(3));
```

Listing 8.1: Exemple d'appel au décorateur générique avec liste des méthodes à cacher

L'idée est donc ici d'automatiser un maximum en décrivant l'utilisation du caching plutot qu'en le programmant directement. On appelle cela la méta-programmation.

La méta-programmation facilite grandement le développement en réduisant le volume de code tout en simplifiant et en rationalisant les programmes.

Dans le langage Java, à partir de sa version 1.5, la méta-programmation est réalisée à l'aide d'annotations.

Pour faire simple, les annotations sont en fait des méta-données associées, entre autres, à des méthodes et interprétées à un moment ou un autre par du code ou par un outil d'analyse.

En définissant une annotation `@Cached` comme renseignant une méthode dont le résultat doit être mis en cache et en l'appliquant à l'exemple de Fibonacci, on obtient le code suivant :

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Cached {

}
```

Listing 8.2: Annotation demandant la mise en cache du resultat d'une methode

```
public class Fibonacci {

  @Cached
  public int fibonacci(int n) {
    if (n < 2)
      return n;
    return (fibonacci(n - 1) + fibonacci(n - 2));
  }

}
```

Listing 8.3: Algorithme simple de la suite de Fibonacci avec annotation demandant la mise en cache du résultat

Le code de l'annotation est très simple, il ne fait que renseigner une demande de mise en cache. Il est lui-même doté de deux méta-annotations qui définissent sa portée et sa rétention. Dans ce cas, l'annotation ne peut être associée qu'à des méthodes et ne sera pas enlevée à la compilation.

Pour sa part, le code de l'algorithme de Fibonacci est inchangé, du moins pour sa partie métier. On s'est juste contenté d'ajouter l'annotation sur la méthode à cacher. L'impact sur le code original est donc minimal.

Par ailleurs, en plus de faciliter la programmation, les annotations ont la particularité de pouvoir être automatiquement intégrées à la javadoc à l'aide de la méta-annotation `@Documented`[1]. L'annotation `@Cached` est modifiée comme suit :

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface Cached {

}
```

Listing 8.4: Annotation demandant la mise en cache du résultat d'une méthode avec documentation

[1] http://java.sun.com/j2se/1.5.0/docs/api/java/lang/annotation/Documented.html

La javadoc ainsi générée est représentée par la figure 8.1 à la page 55.

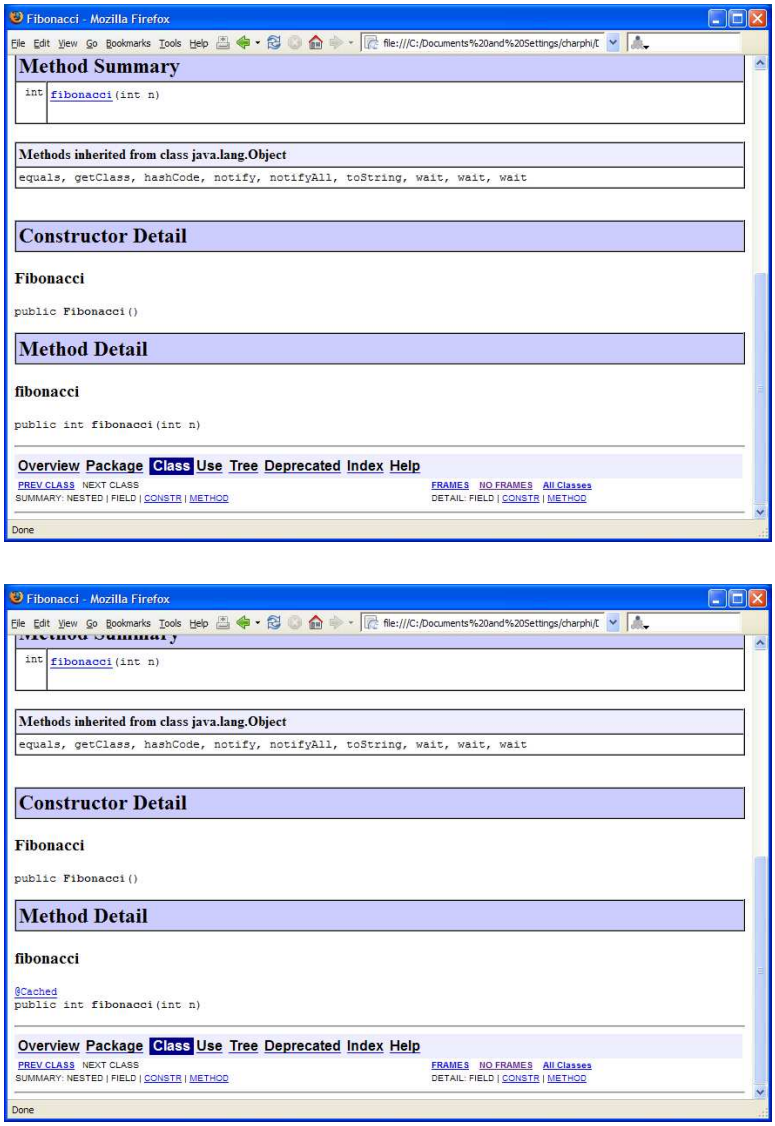

FIG. 8.1: Javadoc sans et avec documentation de l'annotation de caching

Il est aussi possible de consulter les annotations programmatiquement, en utilisant les mécanismes de réflexion de Java [McLaughlin et Flanagan, 2004].

La publicité du caching est donc bien assurée par ces deux mécanismes.

Toutefois, certaines restrictions existent quant à l'utilisation et la déclaration des annotations : elles sont déclarées comme des interfaces mais ne peuvent hériter l'une de l'autre, les méthodes déclarées dans une annotation

ne doivent contenir aucun paramètre et elles doivent en outre ne retourner que des types primitifs. Enfin, les annotations ne peuvent pas contenir de clause `throws`. [Roux, 2004]

8.3 Interception des méthodes annotées

Pour être utiles, les annotations doivent être exploitées par un code dédié.

Dans la précédente section, on avait recours à un décorateur générique. Cette technique a deux limitations importantes : d'une part, le modèle objet traditionnel limite le caching à des méthodes publiques, les méthodes privées étant inaccessibles ; d'autre part, l'interception doit être codée en dur dans l'application et ce pour chaque classe dont on désire cacher les méthodes.

Pour pallier ces défauts, il faut utiliser des solutions qui interceptent ces méthodes annotées à la compilation ou à l'exécution.

Interception statique

Une première solution consiste à compléter le code source à la compilation à l'aide d'une sorte de pré-compilateur.

On pourrait par exemple ajouter une méthode qui possède la même signature que la méthode annotée et qui redirige ses appels vers la véritable méthode qui a été au préable renommée. Appliquée à l'exemple de Fibonacci, cette solution se présente sous cette forme :

```
public class Fibonacci {

  private static Cache cache = CacheManager.getInstance().getCache(
      "fibo");

  @Cached
  public int fibonacci(int n) {
    Integer result = (Integer) cache.get(n);
    if (result == null) {
      result = real_fibonacci(n);
      cache.put(n, result);
    }
    return result;
  }

  public int real_fibonacci(int n) {
    if (n < 2)
      return n;
    return (fibonacci(n - 1) + fibonacci(n - 2));
  }
```

```
}
```

Listing 8.5: Algorithme simple de la suite de Fibonacci avec annotation demandant la mise en cache du résultat et interception par renommage

Ce mécanisme est facilement automatisable et est transparent pour le programmeur et les utilisateurs.

Toutefois, cette solution, bien que simple à comprendre et à mettre en oeuvre n'est pas la meilleure disponible car elle ne permet pas d'être ajoutée dynamiquement à l'exécution.

Interception dynamique

Pour réaliser l'interception dynamique de méthodes, il faut avoir recours à la programmation orientée aspect, ou POA.

La programmation orientée aspect est une nouvelle manière de structurer les applications. Son objectif est de dépasser les limites de la programmation orientée objet traditionnelle.

L'idée principale est d'améliorer la séparation des « préoccupations »(« Separation of Concerns »[Awais et Blair, 2001]) en supprimant les dépendances entre les couches applicatives et les frameworks utilisés. Ces préoccupations forment les différents aspects techniques d'un logiciel, habituellement fortement dépendants entre eux. [Borderie, 2006]

Les avantages de ce paradigme de programmation sont une maintenance plus simple, une modularité plus poussée, une simplification de la programmation et une augmentation de la qualité de code. [Viel, 2005]

Une préoccupation, ou plus simplement fonctionnalité, est réalisée à l'aide d'un aspect et est greffée au code du logiciel par le biais d'un tisseur, un outil permettant d'injecter les codes liés aux aspects dans le code de base d'un programme.

Il existe deux types de tisseurs ; les tisseurs statiques qui agissent à la compilation ou la post-compilation et les tisseurs dynamiques qui appliquent les aspects à chaud pendant l'exécution du programme.

La POA est donc parfaitement adaptée au problème du caching dynamique de méthodes notamment grâce à sa grande modularité.

L'interception de l'annotation de caching définie dans les précédentes sections se fait en deux étapes. On commence par créer un point de jonction qui sert à désigner la partie du code qui sera interceptée. Dans notre cas, il s'agit

de toutes les méthodes qui possèdent l'annotation `@Cached`. Ensuite, on code une méthode concrète qui sera exécutée à chaque interception.

Le code de l'aspect est le suivant :

```
public aspect CachedAspect {

  // definition du point d'interception
  pointcut pc(Cached cached) : execution(* *(..)) && ¶
       @annotation(cached);

  // methode gerant l'interception
  Object around(final Cached cached) : pc(cached) {

    // recuperation du cache
    Cache cache = CacheManager.getInstance().getCache("default");

    // creation de la cle d'acces sur base de la signature
    // de la methode et de la valeur de ses parametres
    String key = thisJoinPoint.getSignature().getName()
        + ((CodeSignature) thisJoinPoint.getSignature())
            .getParameterTypes()
        + Arrays.toString(thisJoinPoint.getArgs());

    // recuperation ou creation de la valeur
    Object value = cache.get(key);
    if (value == null) {
      value = proceed(cached);
      cache.put(key, value);
    }

    // renvoi du resultat
    return value;

  }

}
```

Listing 8.6: Aspect d'interception dynamique de méthodes

On peut constater que le code obtenu n'est pas entièrement en Java. La syntaxe d'un aspect est en effet légèrement différente[2]. On y retrouve la déclaration du point de jonction et le code d'appel au cache entouré d'une déclaration spécifique aux aspects qui permet de définir le type d'interception.

De plus, le code d'appel au cache a été légèrement modifié par rapport au code de l'interception statique. Ceci est dû au fait que cet aspect est générique et ne connait pas à l'avance la méthode qui doit être cachée. Ce qui implique l'ajout d'une technique de création de clé basée sur la signature de la méthode et sur les valeurs de ses paramètres.

Enfin, l'aspect est tissé sur l'application à l'exécution, simplement en ajoutant certains paramètres sur la ligne de commande.

[2] http://www.eclipse.org/aspectj/doc/released/progguide/quick.html

8.4 Développement d'une API concrète

Les précédentes sections de ce chapitre ont démontré l'intérêt de la métaprogrammation par annotation dans le cadre du caching. Toutefois, la solution présentée ne dispose pas de fonctionnalités suffisantes pour être utilisée lors d'un développement. La section courante se propose de bâtir une API concrète.

Classification des méthodes

Si l'on reprend l'exemple du chapitre 6 concernant les requêtes SQL, on peut distinguer deux types de méthodes ; celles qui produisent un résultat sans modifier quoi que ce soit (les sélections), et celles qui effectuent des actions (les insertions et mises à jour). Elles sont respectivement baptisées méthodes de lecture et méthodes d'écriture.

Les premières voient leurs résultats mis en cache, les secondes invalident les données du cache et ne peuvent être ignorées sous peine d'entraîner une incohérence entre les données contenues dans le cache et les données réelles.

Ainsi, la précédente annotation, `@Cached`, est remplacée par deux nouvelles annotations qui sont respectivement `@ReadMethod` et `@WriteMethod`. En voici les déclarations :

```
@Documented
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface ReadMethod {

}
```

Listing 8.7: Annotation de méthodes de lecture

```
@Documented
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface WriteMethod {

}
```

Listing 8.8: Annotation de méthodes d'écriture

Il aurait été possible de garder la première annotation en lui ajoutant un paramètre booléen ou une énumération. Toutefois, ce choix n'a pas été retenu car, d'une part, les deux annotations divergent dans la suite de cette section et, d'autre part, l'héritage n'existe pas pour les annotations.

La classe suivante, dédiée à la manipulation d'une liste d'utilisateurs, illustre l'intérêt de ces deux nouvelles annotations. On y retrouve des méthodes de type manipulation et de type consultation.

Son code annoté est le suivant :

```
public class UserDAO {

  protected Map<Long, User> store;

  public UserDAO(Map<Long, User> store) {
    this.store = store;
  }

  @WriteMethod
  public void add(User user) {
    store.put(user.getId(), user);
  }

  @WriteMethod
  public void update(User user) {
    store.put(user.getId(), user);
  }

  @WriteMethod
  public void remove(User user) {
    store.remove(user.getId());
  }

  @WriteMethod
  public void clear() {
    store.clear();
  }

  @ReadMethod
  public User getById(long id) {
    return store.get(id);
  }

  @ReadMethod
  public Collection<User> list() {
    return store.values();
  }
}
```

Listing 8.9: Classe de gestion d'une liste d'utilisateurs

On constate bien que les méthodes de manipulation (`add`, `update`, `remove`, `clear`) sont annotées en écriture et les méthodes de consultation (`getById`, `list`) sont annotées en lecture.

Identification des caches

Il n'est pas rare que l'on doive utiliser plusieurs caches différents dans un même programme. Les données cachées peuvent, par exemple, être de types différents ou requérir des manipulations particulières plus adaptées à un système qu'à un autre.

Il est donc nécessaire de nommer le cache que l'on désire utiliser. Le code est adapté comme suit :

```
@Documented
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface ReadMethod {

  String cacheName() default "default";

}
```

Listing 8.10: Annotation de méthodes de lecture avec identification du cache à utiliser

```
@Documented
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface WriteMethod {

  String cacheName() default "default";

}
```

Listing 8.11: Annotation de méthodes d'écriture avec identification du cache à utiliser

La définition d'un nom de cache par défaut n'est là que pour simplifier le code dans le cas où un seul cache est nécessaire. Le choix d'un cache personnalisé se fait de la manière suivante :

```
  @ReadMethod(cacheName = "usercache")
  public User getById(long id) {
    return store.get(id);
  }
```

Listing 8.12: Identification d'un cache

Génération des clés de lecture

Dans la section précédente consacrée à l'interception des méthodes annotées, la version dynamique a été réalisée à l'aide d'un mécanisme générique de création de clé basé sur la signature et les paramètres de la méthode à cacher.

Ce type de solution est tout à fait fonctionnel mais aussi parfois excessif ; dans l'exemple de la manipulation d'utilisateurs, la clé d'un enregistrement dans le cache pourrait être soit le numéro d'indentifiant de l'utilisateur, ici un entier, soit le nom de la méthode `list`.

De plus, il n'est pas exclu que deux méthodes différentes produisent les mêmes données. Il serait judicieux d'éviter les doublons.

Ce problème peut être résolu en spécifiant, dans le code source, la classe qui se chargera de générer la clé de stockage d'un enregistrement dans le cache.

L'annotation `@ReadMethod` est modifiée de la sorte :

```
@Documented
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface ReadMethod {

  String cacheName() default "default";

  Class<? extends IKeyGen> keygen() default DefaultBehavior.class;

}
```

Listing 8.13: Annotation de méthodes de lecture avec identification du cache à utiliser et génération des clés de lecture

La personnalisation de la clé est réalisée à l'aide d'une implémentation de l'interface `IKeyGen`. De nouveau, une solution par défaut est prévue pour faciliter le travail du développeur.

L'interface `IKeyGen` est définie comme suit :

```
public interface IKeyGen {

  public String getKey(ICacheContext context);

}
```

Listing 8.14: Générateur de clés de lecture

L'unique paramètre de la méthode `getKey` sert simplement à regrouper toutes les informations nécessaires à la génération de la clé ; que ce soit l'objet ciblé, la méthode ou les paramètres d'appel.

```
public interface ICacheContext {

  public Object getTarget();

  public Method getMethod();

  public Object[] getParameters();
```

```
}
```

Listing 8.15: Informations relatives à l'appel au cache

Enfin, en appliquant ce système à l'exemple de la gestion d'une liste d'utilisateurs, on obtient une classe qui crée une clé à partir de l'identifiant de l'utilisateur. Il suffit alors d'ajouter l'option dans le code de la classe `UserDAO`.

```
public class UserKeyGen implements IKeyGen {

  public String getKey(ICacheContext context) {
    // recuperation du nom de la methode
    // pour determiner le comportement
    String method = context.getMethod().getName();

    // chaine 'list' comme cle
    if (method.equals("list"))
      return "list";

    // cle a partir de l'id de l'utilisateur
    if (method.equals("getById")) {
      if (context.getParameters()[0] instanceof User) {
        User user = (User) context.getParameters()[0];
        return Long.toString(user.getId());
      }
    }

    return null;
  }

}
```

Listing 8.16: Générateur de clés des utilisateurs

```
public class UserDAO {

  protected Map<Long, User> store;

  public UserDAO(Map<Long, User> store) {
    this.store = store;
  }

  @WriteMethod
  public void add(User user) {
    store.put(user.getId(), user);
  }

  @WriteMethod
  public void update(User user) {
    if (store.containsKey(user.getId()))
      store.put(user.getId(), user);
  }

  @WriteMethod
  public void remove(User user) {
    store.remove(user.getId());
  }
```

```
  @WriteMethod
  public void clear() {
    store.clear();
  }

  @ReadMethod(keygen = UserKeyGen.class)
  public User getById(long id) {
    return store.get(id);
  }

  @ReadMethod(keygen = UserKeyGen.class)
  public Collection<User> list() {
    return store.values();
  }
}
```

Listing 8.17: Classe de gestion d'une liste d'utilisateurs avec génération des clés de lecture

Invalidation ciblée

L'invalidation effectuée par l'annotation `@WriteMethod` est absolue dans le sens où elle invalide tout aveuglément. Il serait pourtant plus intéressant de n'invalider que les données qui ont été modifiées.

Par exemple, un ajout modifie la liste des utilisateurs tandis qu'une mise à jour modifie un utilisateur bien précis ainsi que cette liste.

Il faudrait donc pouvoir cibler une invalidation de trois façons ; invalider tout le cache, invalider une liste précise de clés et enfin invalider des ensembles de clés.

Pour ce faire, on associe les clés à un ou plusieurs groupes nommés. L'invalidation d'un groupe doit entrainer l'invalidation de toutes les clés de ce groupe. Par contre, l'invalidation d'une clé n'a aucune incidence sur le ou les groupes auquels elle appartient.

L'association d'une clé à un groupe est obtenue par l'ajout de la propriété `groups` à la définition de l'annotation de lecture :

```
@Documented
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface ReadMethod {

  String cacheName() default "default";

  Class<? extends IKeyGen> keygen() default DefaultBehavior.class;

  Class<? extends IGroupsGen> groups() default DefaultBehavior.class;
```

```
}
```

Listing 8.18: Annotation de méthodes de lecture avec identification du cache à utiliser génération des clés de lecture et invalidation ciblée

Cette propriété est définie par l'interface `IGroupsGen`. Celle-ci contient une unique méthode qui renvoie un tableau de chaînes de caractères identifiants des groupes sur base du contexte de la méthode. Son code est le suivant :

```
public interface IGroupsGen {

  public String[] getGroups(ICacheContext context);
}
```

Listing 8.19: Générateur de groupes pour l'invalidation ciblée

L'implémentation par défaut de cette interface se contente de renvoyer un tableau vide et a donc pour conséquence de n'associer une clé à aucun groupe.

Du coté de l'annotation d'écriture, nous avons besoin de deux nouveaux éléments ; d'une part, une énumération qui détermine le type d'invalidation, et d'autre part une interface qui renvoie la liste des clés ou des ensembles de clés à invalider. Le code de l'annotation d'écriture est complété comme suit :

```
@Documented
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface WriteMethod {

  String cacheName() default "default";

  WriteType type() default WriteType.ALL;

  Class<? extends ITargeter> targeter() default ¶
       DefaultBehavior.class;

}
```

Listing 8.20: Annotation de méthodes d'écriture avec identification du cache à utiliser et invalidation ciblée

La première propiété, `type`, représente le type d'invalidation et est définie par l'énumération suivante :

```
public enum WriteType {

  ALL,    // invalide tout le cache
  KEYS,   // invalide une liste de cles
  GROUPS // invalide des ensembles de cles

}
```

Listing 8.21: Types d'invalidation ciblée

La seconde propriété, `targeter`, désigne l'interface qui permet d'obtenir les informations nécessaires à l'invalidation. Cette interface contient une méthode qui renvoie une liste de clés ou de groupes en fonction du type d'invalidation.

```
public interface ITargeter {

  public String[] getTargets(ICacheContext context, WriteType type);

}
```

Listing 8.22: Interface définissant les cibles d'invalidation

La mise en place de l'invalidation ciblée dans l'exemple de la gestion d'une liste d'utilisateur consiste en l'implémentation de l'interface définissant les cibles d'invalidation, ainsi qu'en l'ajout de l'option correspondante dans la classe `UserDAO`.

```
public class UserTargeter implements ITargeter {

  public String[] getTargets(ICacheContext context, WriteType type) {
    if (type.equals(WriteType.KEYS)) {
      if (context.getParameters()[0] instanceof User) {
        User user = (User) context.getParameters()[0];
        String id = Long.toString(user.getId());
        // cible un utilisateur ainsi que la liste
        return new String[] { id, "list" };
      }
    }
    return new String[0];
  }

}
```

Listing 8.23: Cibles d'invalidation pour les utilisateurs

L'exemple n'emploie que l'invalidation complète et l'invalidation ciblée de clés. Les clés d'invalidations sont le numéro d'identification de l'utilisateur que l'on modifie, supprime ou ajoute et la liste des utilisateurs.

```
public class UserDAO {

  protected Map<Long, User> store;

  public UserDAO(Map<Long, User> store) {
    this.store = store;
  }

  @WriteMethod(type = WriteType.KEYS, targeter = UserTargeter.class)
  public void add(User user) {
    store.put(user.getId(), user);
  }

  @WriteMethod(type = WriteType.KEYS, targeter = UserTargeter.class)
  public void update(User user) {
```

```
    if (store.containsKey(user.getId()))
      store.put(user.getId(), user);
  }

  @WriteMethod(type = WriteType.KEYS, targeter = UserTargeter.class)
  public void remove(User user) {
    store.remove(user.getId());
  }

  @WriteMethod
  public void clear() {
    store.clear();
  }

  @ReadMethod(keygen = UserKeyGen.class)
  public User getById(long id) {
    return store.get(id);
  }

  @ReadMethod(keygen = UserKeyGen.class)
  public Collection<User> list() {
    return store.values();
  }
}
```

Listing 8.24: Classe de gestion d'une liste d'utilisateurs avec génération des clés de lecture et invalidation ciblée

Validation avancée

Tel quel, le système basé sur les deux annotations n'est pas suffisant pour garantir la validité d'une donnée dans le cache. En effet, il est possible qu'un serveur change son état sans en avertir les différents caches qui s'y rapportent. C'est le scénario typique d'un schéma client-serveur où plusieurs clients différents peuvent invoquer les services d'un serveur d'application. [Pfeifer et Jakschitsch, 2003]

Il existe plusieurs solutions pour gérer ce problème. On peut par exemple étendre le gestionnaire de cache pour qu'il s'enregistre auprès du serveur et soit ainsi tenu au courant des éventuelles modifications. Une autre solution consiste à n'utiliser qu'un seul cache situé sur le serveur. On peut aussi avoir recours à un serveur d'invalidation qui coordonne les invalidations entre tous les clients.

La solution proposée ici consiste à valider la donnée récupérée dans le cache. Cette validation peut être réalisée sur base du contenu de la donnée ou sur base d'informations extérieures.

Par exemple, une page web contient dans son code source sa date d'expiration. Il est donc possible de déterminer sa validité en comparant cette date d'expiration avec la date courante.

Concrètement, la validation avancée est gérée à l'aide d'un nouveau paramètre de l'annotation de lecture qui détermine la classe qui valide la donnée contenue dans le cache. Cette classe implémente l'interface suivante :

```
public interface IValidator {

  public boolean isValid(ICacheContext context, Object value);

}
```

Listing 8.25: Interface de validation avancée

La méthode `isValid` valide un objet passé en paramètre. Elle dispose du contexte de l'interception.

Il ne reste plus qu'à compléter l'annotation de lecture pour tenir compte de cette validation. De nouveau, dans un soucis de simplicité, une classe par défaut est spécifiée. Celle-ci se contente de valider positivement toutes les données qu'on lui soumet. Le code de l'annotation de lecture est le suivant :

```
@Documented
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface ReadMethod {

  String cacheName() default "default";

  Class<? extends IKeyGen> keygen() default DefaultBehavior.class;

  Class<? extends IGroupsGen> groups() default DefaultBehavior.class;

  Class<? extends IValidator> validator() default ¶
      DefaultBehavior.class;
}
```

Listing 8.26: Annotation de méthodes de lecture avec identification du cache à utiliser génération des clés de lecture invalidation ciblée et validation avancée

La mise en cache d'un flux RSS sur Internet illustre une des possibilités de cette validation :

```
public class WireFeedExample {

  /**
   * Recupere un feed (rss ou atom) sur le net
   */
  @ReadMethod(validator = WireFeedValidator.class)
  public WireFeed getFeed(URL url) throws FeedException,
      IOException {
```

```
    XmlReader reader = new XmlReader(url);
    WireFeedInput input = new WireFeedInput();
    return input.build(reader);
  }

}
```

Listing 8.27: Mise en cache d'un flux rss

La méthode `getFeed` renvoie un flux RSS sur base de son url. Cette méthode est accompagnée par l'annotation de lecture pour laquelle on a renseigné une classe de validation particulière. Celle-ci se contente d'exploiter une information contenue dans le flux RSS. Il s'agit de sa date d'expiration. De même que pour l'exemple de la page web présenté plus haut, on détermine la validité de l'objet par une simple comparaison de dates. Le code de validation est le suivant :

```
public class WireFeedValidator implements IValidator {

  public boolean isValid(ICacheContext context, Object value) {
    if (value instanceof Channel) {
      Channel channel = (Channel) value;

      // recuperation de la date de publication et du ttl
      Date pubDate = (channel.getPubDate() != null) ? channel
          .getPubDate() : channel.getLastBuildDate();
      int ttl = channel.getTtl() * 60 * 1000;

      // calcul de la date d'expiration
      GregorianCalendar calendar = new GregorianCalendar();
      calendar.setTime(pubDate);
      calendar.add(Calendar.MILLISECOND, ttl);
      Date expDate = calendar.getTime();

      // date courante
      Date now = new Date();

      // valide si date d'expiration > date courante
      return expDate.after(now);
    }
    return true;
  }

}
```

Listing 8.28: Code de validation d'un flux rss

Il est ainsi possible de valider un objet sur base de son contenu mais pas seulement. En effet, rien n'empêche le programmeur d'avoir recours à des appels externes dans son code de validation. Une grande liberté d'action lui est laissée.

Résumé des processus de lecture et d'écriture

La gestion d'un caching dynamique de méthodes tel que développé ici peut être résumé en deux grands processus distincts : les processus de lecture et d'écriture.

Le processus de lecture, représenté par la figure 8.2 à la page 71, commence par récupérer l'annotation de la méthode que l'on est en train d'intercepter. Cette annotation sert à déterminer les informations nécessaires au traitement que sont le cache à utiliser, la clé identifiant la valeur à retrouver dans le cache et les groupes auxquels appartient cette clé.

Ensuite, la valeur est récupérée dans le cache. Si cette valeur n'est pas `null`, on la valide à l'aide de la classe renseignée par l'annotation de lecture. Si le résultat de cette validation est positif, on retourne la valeur.

Par contre, si la valeur est `null`, c'est-à-dire qu'elle n'est pas stockée dans le cache, ou si la valeur n'est plus valide, il faut alors la recréer, la sauvegarder dans le cache et enfin associer sa clé avec ses groupes en vue d'une éventuelle invalidation postérieure.

Pour sa part, le processus d'écriture, représenté par la figure 8.3 à la page 72, récupère l'annotation de la méthode interceptée pour dans un premier temps déterminer le cache à utiliser. Ensuite, selon le type d'invalidation défini par l'annotation, le processus va soit invalider entièrement le cache, soit invalider des clés bien précises, ou encore invalider des groupes de clés. Enfin, le code métier est exécuté.

8.5 Limites et évolutions possibles

Bien qu'apportant de grandes améliorations, le caching dynamique de méthodes n'est pas sans faille.

En effet, le recours à la programation orientée aspect apporte les désavantages de cette technique. Le code résultant d'un tissage est ainsi plus difficile à analyser lors des phases de débogage. On peut se demander aussi comment éviter de tisser plusieurs fois un même code.

Par ailleurs, la cohérence des données est assurée par la définition des dépendances entre les méthodes. La qualité de cette définition est donc primordiale. Or, une mise en oeuvre naïve d'un cache cohérent peut résulter en une solution inefficace. Dans certains cas, la gestion cohérente d'un cache n'est pas orthogonale à l'application. Enfin, la prise en compte d'un cache cohérent n'est pas forcément trivial.[Bouchenak *et al.*, 2005]

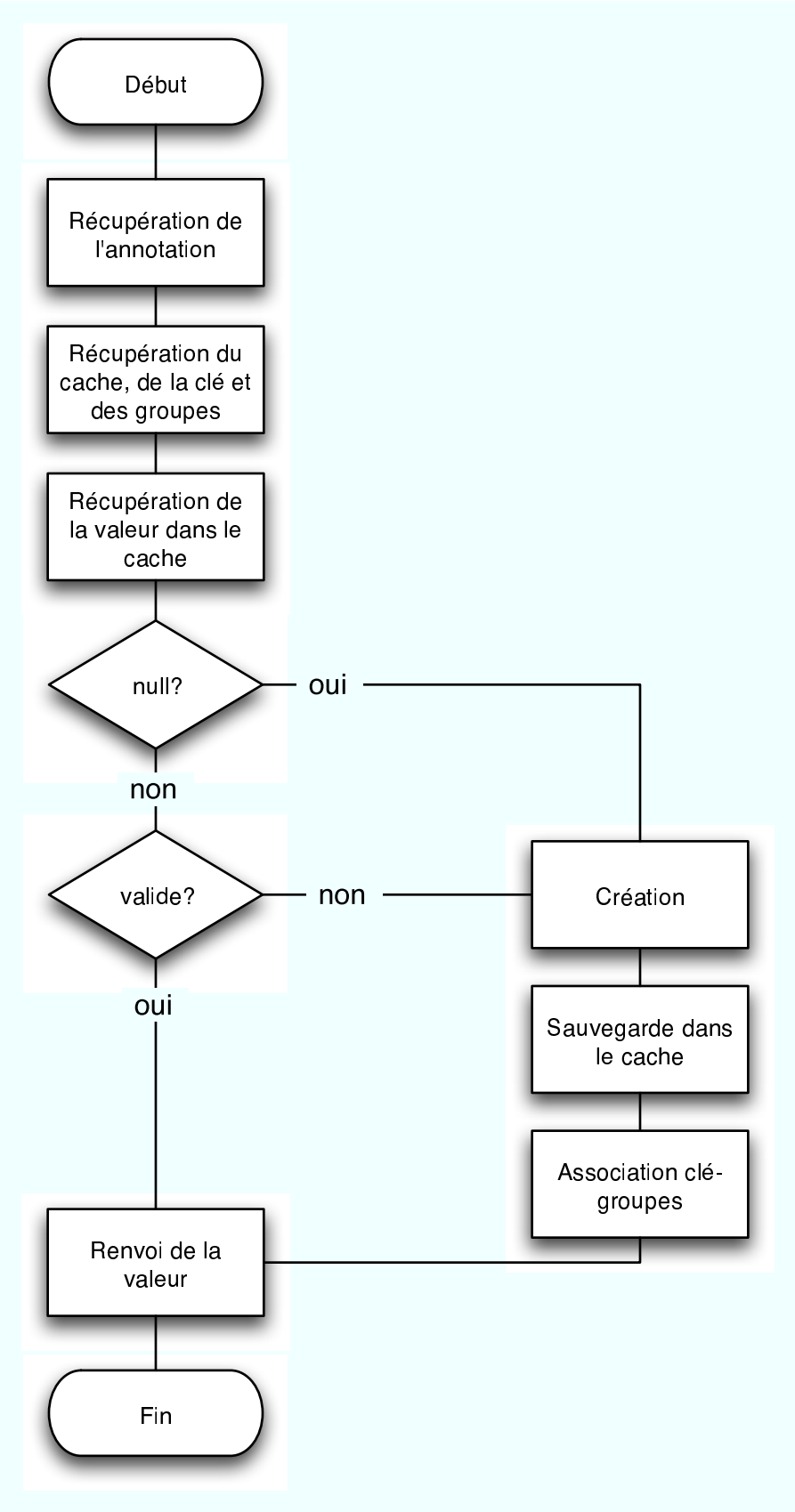

FIG. 8.2: Diagramme de processus de lecture

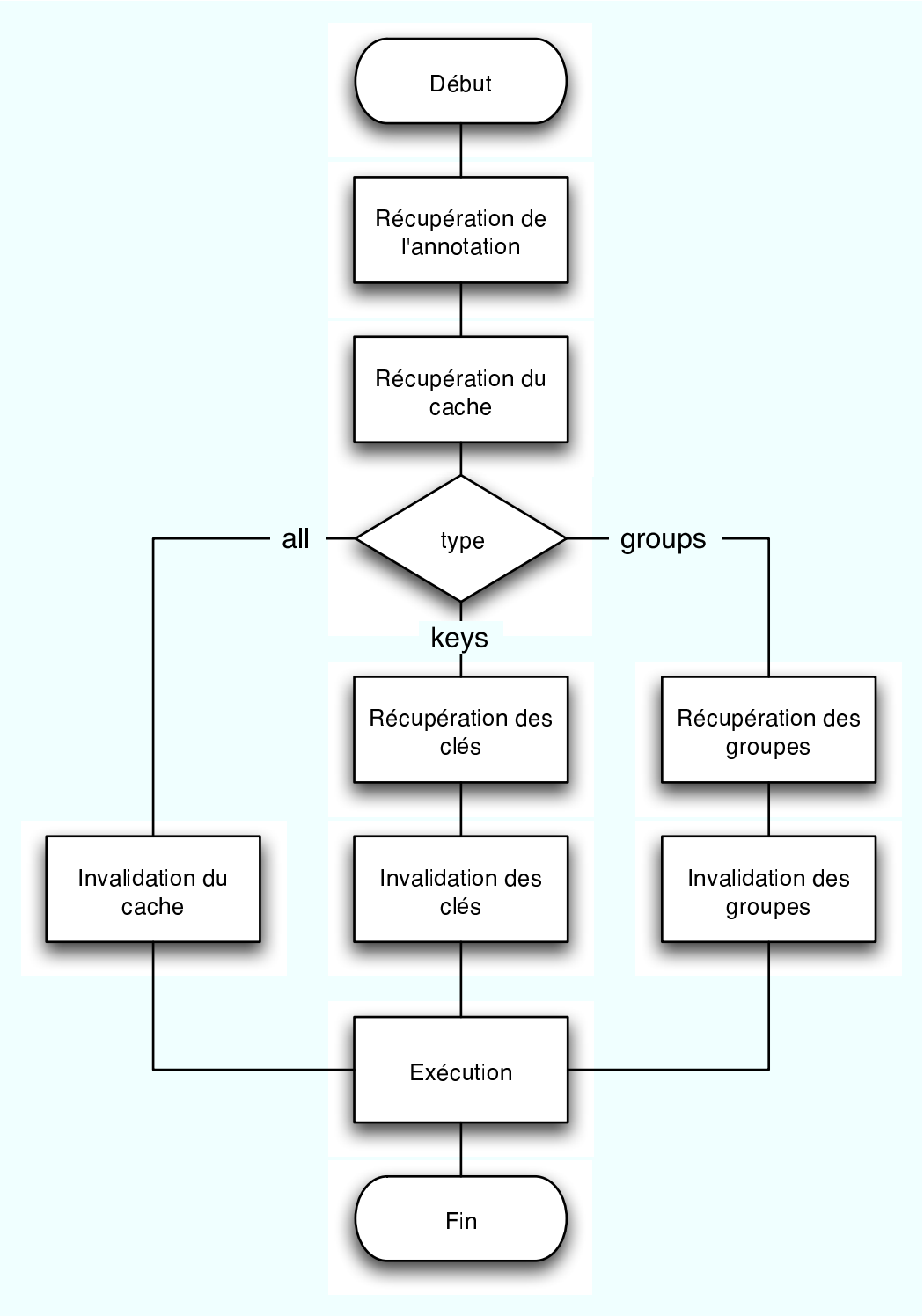

FIG. 8.3: Diagramme de processus d'écriture

De plus, la flexibilité des annotations est assez faible. Outre le fait qu'il ne soit pas possible d'utiliser l'héritage, les types de données que l'on peut passer en paramètre à une annotation sont limités. Il est par exemple impossible de passer l'instance d'une classe.

Il est également difficile d'empêcher certaines erreurs flagrantes de programmation comme la définition d'une annotation de lecture et d'écriture sur une même méthode.

Heureusement, les prochaines versions de Java intègreront un outil spécialisé dans le contrôle de contraintes sémantiques directement dans le compilateur.[Zhicheng-Li, 2006]

De même, ces nouvelles techniques sont de plus en plus utilisées dans les différentes bibliothèques écrites en Java comme par exemple la troisième version des EJB. Les éditeurs s'adaptent au changement et on voit ainsi apparaître des débogueurs capables de gérer la programmation orientée aspect.

En outre, l'utilisation de caching déclaratif élimine les dépendances du code à l'implémentation du cache, ce qui clarifie la séparation des responsabilités, augmente la modularité du code et offre la possibilité de postposer certaines décisions sur l'implémentation du cache.[Ruiz, 2006]

Enfin, le caching dynamique de méthodes apporte une solution élégante et efficace au problème de l'intégration dans le processus de développement. Les programmeurs peuvent ainsi se concentrer sur le code métier de leur application. Cette intégration peut même être réalisée dans les dernières étapes du cycle de développement. Les différentes embûches mises en évidence dans les précédents chapitres sont pratiquements résolues. Il reste quelques difficultés dues à un certain manque de maturité des différentes technologies employées mais ces difficultés devraient être résolues ou contournées dans un futur proche.

8.6 Travaux connexes

L'association entre programmation orientée aspect et caching n'est pas courante mais a fait l'objet de quelques études.

Le caching de données statiques est souvent cité comme exemple d'utilisation de la programmation orientée aspect. Une étude menée par l'INRIA (INRIA - AOP-Based Caching of Dynamic Web Content : Experience with J2EE Applications)[Bouchenak *et al.*, 2005] vise à déterminer la possibilité d'aspectiser un cache cohérent de documents web générés dynamiquement.

La solution employée se veut totalement transparente en automatisant la gestion de la cohérence des données et est réalisée à l'aide d'un mécanisme qui intercepte et analyse les servlets lors de leur exécution.

Les conclusions de cette étude sont que la gestion cohérente d'un cache de documents dynamiques n'est pas orthogonale à l'application. La prise en compte d'un cache cohérent de documents web dynamique en utilisant la programmation par aspect s'est avérée ne pas être triviale.

Une autre étude (Method-Based Caching in Multi-Tiered Server Application) réalisée à l'université de Karlsruhe en Allemagne [Pfeifer et Jakschitsch, 2003] tente de généraliser le problème au modèle client-serveur. Le système imaginé ici ne se limite plus aux servlets mais s'étend aux méthodes des classes.

De même que pour mbc, les méthodes sont divisées en deux groupes que sont les méthodes read et write. L'identification de ces méthodes est réalisée à l'aide de fichiers de configuration xml.

Pour sa part, la cohérence des données est spécifiée à l'aide d'un modèle descriptif. Ce modèle décrit les dépendances entre les méthodes. De nouveau, le principal problème rencontré par cette solution se situe au niveau de la cohérence des données.

Un autre problème non envisagé par cette étude vient de la verbosité du xml. En effet, la description des dépendances s'avère assez fastidieuse. De plus, cette description est "déconnectée" du code métier. On retrouve ici les problèmes habituels de la version 2 des EJB. Une version plus évoluée tente cependant de résoudre ce problème à l'aide d'annotations mais reste trop limitée.

Une autre solution vient d'une librairie de caching du framework Spring (Declarative Caching Services for Spring)[Ruiz, 2006]. Cette solution utilise à la fois des fichiers de configuration xml et des annotations pour intercepter à l'exécution des appels à des méthodes. Elle supporte aussi l'invalidation ciblée ou totale.

Toutefois, l'expressivité de ces annotations est réduite au maximum ; il n'est ainsi possible que de renseigner un identifiant de configuration xml. De plus, cette libraire n'est utilisable que dans le cadre du framework Spring et a donc un intérêt limité.

Troisième partie

Cas pratiques

CHAPITRE 9

PROTOCOLE DE TEST

La troisième et dernière partie de ce mémoire porte sur la mise en pratique des concepts évoqués dans les deux premières parties. Il s'agit ici d'évaluer l'efficacité de l'optimisation par caching, que ce soit au niveau de la facilité de développement ou de la performance pure.

Deux applications seront étudiées dans cette partie : une simple et une complexe. La première est basée sur l'application FTB décrite dans le chapitre du scénario d'intégration. La seconde porte sur une application d'analyse statistique développée en interne à la Banque Nationale de Belgique et réalisée en C#.

Chaque étude sera divisée en quatre sections contenant :

- une brève description de l'application étudiée,
- la manière dont l'intégration a été réalisée,
- une analyse des performances obtenues avec et sans cache,
- quelques constatations déduites de l'expérience acquise.

9.1 Implémentation du caching

La solution concrète de caching choisie pour cette étude est l'implémentation directe de l'API construite dans le chapitre consacré au caching dynamique de méthodes.

Cette implémentation est nommée mbc pour Method Based Caching et est divisée en cinq packages :

- jcache : les interfaces du JSR-107
- jcacheimpl : une implémentation de jcache
- mbc : les interfaces et les classes définissant mbc
- mbc-jcache : une implémentation de mbc à l'aide de jcache
- mbc-apsectj : un aspect permettant l'interception des méthodes annotées

Les dépendances entre les packages sont les suivantes :

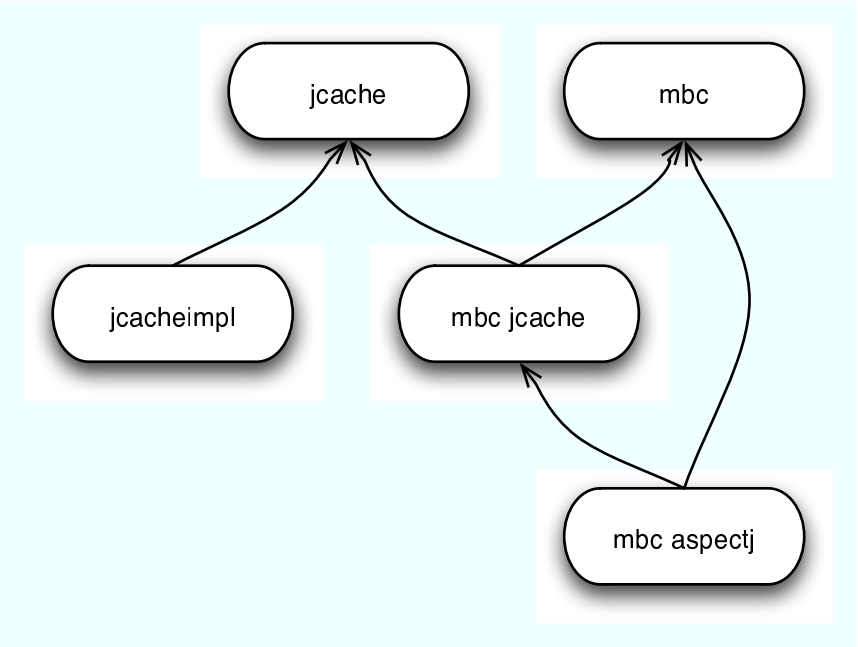

FIG. 9.1: Diagramme de dépendances entre les packages de mbc

Les packages sont agencés de façon à magnifier la modularité du framework. Il est ainsi possible d'avoir recours à n'importe quelle implémentation de jcache ou n'importe quelle technologie de programmation orientée aspect. D'ailleurs, le système tel que conçu ici ne requiert pas forcément l'utilisation de jcache. On aurait pu s'en passer. Seul le package mbc est indispensable.

Enfin, le framework dispose d'un mode de fonctionnement particulier dans lequel il compare les données mises en cache aux données calculées lors d'un appel à une méthode cachée. Ceci permet de trouver les erreurs éventuelles dans la description des dépendances entre les méthodes.

9.2 Mesure des performances

Pour déterminer l'efficacité de la solution de caching, il faut avoir recours à une analyse des performances. Dans le cadre d'un cache, cette mesure est assez particulière.

Habituellement, en informatique, on a recours à la théorie de la complexité pour déterminer les performances d'un algorithme. Cette théorie se base en partie sur le cas le plus mauvais. Or, un cache ne peut garantir la récupération d'une donnée, ce qui implique que la complexité est toujours maximale.

La théorie de la complexité n'est donc pas capable d'apporter des informations sur l'efficacité du caching. La seule solution disponible est la mesure directe du temps d'exécution d'une méthode avec et sans cache.

Cette évaluation est réalisée à l'exécution, sur un jeu de tests ou sur une utilisation courante et est enregistrée dans des fichiers de logs qui permettent une analyse postérieure. Il est ainsi possible de générer diverses statistiques sur le comportement du cache.

Concrètement, un fichier de logs est complété à l'exécution par de simples lignes de texte dont le format des enregistrements est le suivant :

- Nom canonique de la classe
- Signature de la méthode
- Temps d'exécution en nanosecondes
- Nom du cache
- Clé utlisée
- Liste des groupes
- Etat de la donnée (valide, invalide ou manquante)

Il aurait été possible d'avoir recours à la technologie des MBeans pour analyser à chaud le comportement du framework. Ce choix n'a pas été retenu car la deuxième application analysée est écrite en C#, tourne sur le framework .NET et n'est donc pas compatible.

CHAPITRE 10

APPLICATION SIMPLE

Le programme étudié dans ce chapitre est une implémentation de l'application FTB décrite dans le chapitre sur le scénario d'intégration. Le but est d'étudier la variation des performances en fonction des paramètres de configuration du système de cache.

En ce qui concerne la réalisation, l'application FTB est entièrement réalisée en Java. La figure 10.1 de la page 80 montre deux captures d'écran issues de FTB. La première image correspond à un tableau affichant la liste des catégories accompagnées du nombre de fichiers qu'elles contiennent et de l'espace qu'elles occupent sur le disque. La seconde image représente la proportion d'espace disque utilisé par chaque catégorie.

10.1 Intégration

L'utilisation du cache dans l'application FTB est très simple et est localisée sur une seule méthode : il s'agit de la méthode qui renvoie la catégorie d'un fichier sur base de son extension.

Au niveau de l'intégration dans le processus de développement, l'utilisation du framework de caching dynamique de méthode a permis une intégration tardive. Celle-ci a eu lieu dans les dernières étapes et a consisté en l'ajout d'une simple annotation dans le code source et de quelques paramètres à la commande d'exécution.

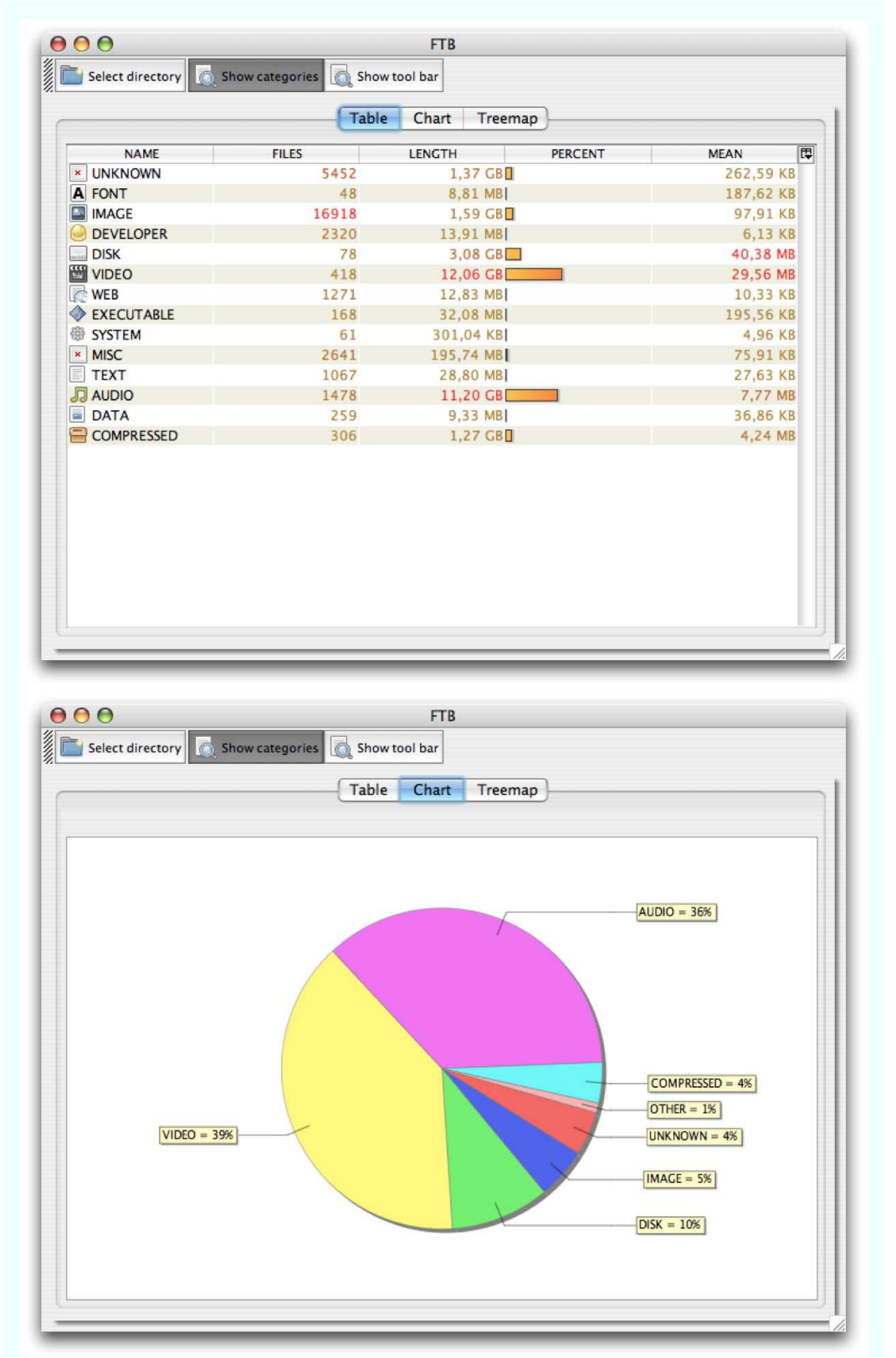

FIG. 10.1: Captures d'écran de l'application FTB

10.2 Performances

Comme expliqué dans la première partie, les performances d'une solution de caching dépendent des paramètres encodés. Ainsi, moins on réserve de place au cache, moins on consomme de mémoire. Par contre, le cache devra avoir recours plus souvent à l'éviction. Il convient donc de trouver un bon compromis entre les deux.

Le tableau suivant (tab. 10.1, p 81) montre l'impact des paramètres de la solution sur les performances. Les données sont issues de trois configurations différentes : la première porte sur un cache qui ne mémorise aucune valeur, la deuxième, sur un cache mémoire qui mémorise un maximum de 10 valeurs et la troisième, sur un cache mémoire qui mémorise un maximum de 100 valeurs. Les fichiers analysés sont au nombre de 30655. La méthode d'éviction choisie est la méthode LRU, les types de fichiers étant assez homogènes dans un même répertoire.

Type de cache	Durée moyenne	Durée totale	% Valide
Null	22 ms	684 s	0 %
Mémoire (10)	962 μs	29 s	96 %
Mémoire (100)	312 μs	9 s	99 %

TAB. 10.1: Impact des paramètres sur les performances de FTB

En analysant ce tableau, on constate qu'une simple utilisation d'un cache mémoire limité à 10 divise le temps d'exécution par 23. Le passage de la limite de mémoire à 100 permet quant à lui de diviser ce temps par 69.

Le gain n'est donc pas linéaire par rapport à l'espace mémoire octroyé au cache. Même en réservant une place suffisante pour contenir toutes les données, le programme est toujours limité par le temps de récupération d'une donnée dans le cache ainsi que son ajout initial.

Par ailleurs, en analysant plus profondément les fichiers de log (fig. 10.2, p. 82), on s'aperçoit que le temps moyen d'exécution de données invalides ou manquantes augmente avec la taille du cache. Ceci est dû au problème évoqué dans le chapitre consacré au temps réel : l'implémentation du cache utilise la `HashMap` habituelle du JRE qui ne garantit malheureusement pas un temps d'accès constant quelque soit la taille des données qu'elle contient.

Le graphique 10.3 de la page 83 représente l'évolution du taux de validité des données dans le cache, c'est à dire le rapport entre le nombre de données valides par rapport au nombre total de requêtes. Dans cet exemple, la configuration mémoire 100 atteint plus rapidement sa limite que la version 10. Sa

FIG. 10.2: Temps d'exécution moyen d'une requête invalide ou manquante

durée totale y est bien sûr plus courte. De même, son taux de validité est plus important.

10.3 Constatations

L'utilisation de caching a permi d'améliorer les performances de manière significative et ce à moindre frais. En effet, les performances sont multipliées par un facteur de 23 ou 69 selon la configuration et l'intégration elle-même ne pourrait pas être plus simple.

Toutefois, il faut reconnaître qu'il aurait été possible de faire mieux en remaniant l'algorithme de recherche des catégories en fonction de l'extension, par exemple en triant au préalable toutes les extensions pour n'effectuer qu'une seule requête par groupe d'extensions.

Le caching a ici le gros avantage de laisser le code métier intact.

En outre, les performances ne sont pas améliorables à l'infini. Elles peuvent en plus être variables en fonction du type de données analysées. Par exemple, un répertoire de vidéos contient plus de fichiers de même type qu'un répertoire système et a donc un meilleur rendement dans le cache.

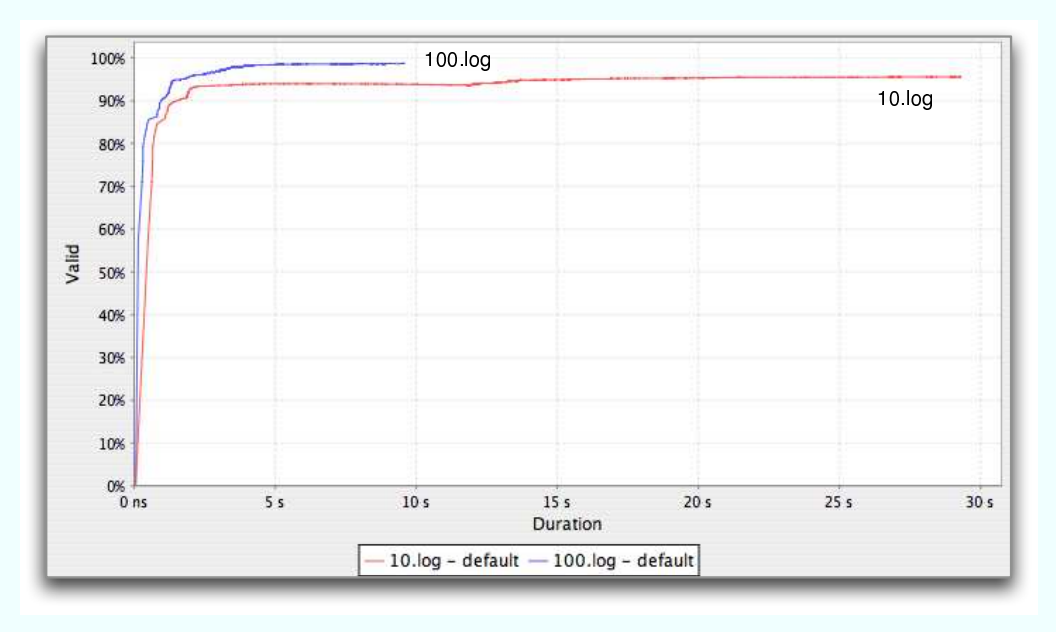

FIG. 10.3: Evolution du taux de validité des données dans le cache de FTB

CHAPITRE 11

APPLICATION COMPLEXE

L'application d'analyse des courbes synthétiques, nommée CS, calcule les indicateurs et les courbes synthétiques des enquêtes de conjoncture de la Banque Nationale de Belgique. Elle comporte plusieurs dizaines de milliers de lignes de code et a été réalisée sur une période de un an.

Ce programme récupère les résultats de lignes de synthèse et permet de définir des indicateurs synthéthiques comme des combinaisons linéraires de lignes de synthèse ainsi que des courbes synthétiques comme des combinaisons linéraires d'indicateurs synthétiques. Plusieurs méthodes d'agrégation, de désaisonalisation et de lissage peuvent être associées à chaque indicateur et courbe synthétique. Enfin, l'application CS permet l'archivage des résultats produits. Ces résultats sont associés à des codes de publication pour la production des parutions en version papier et électronique.

Les lignes de synthèse se définissent comme l'agrégat des résultats de premières globalisations de plusieurs produits et/ou d'autres résultats de lignes de synthèse. A l'instar des premières globalisations, les lignes de synthèse fournissent des résultats de réponses discrètes ou continues.

Les lignes de synthèse fournissent des résultats :

- au sein d'un même groupe d'enquêtes appelé secteur,
- pour une période,
- pour une zone géographique appelée région,
- avec ou sans partition,

– avec ou sans catégorie de biens,
– pondérés (par un critère) ou non,
– selon une méthode d'agrégation (somme ou moyenne pondérée),
– selon un code de première globalisation.

Un indicateur synthétique est une moyenne pondérée ou non de résultats de lignes de synthèse. Les indicateurs synthétiques sont caractérisés par :

– un code,
– une zone géographique appelée région,
– une pondération éventuelle,
– une fréquence, c'est-à-dire la périodicité de collecte des données,
– un secteur,
– une composition pondérée de lignes de synthèse ;
– une méthode de calcul (désaisonalisation et/ou lissage).

Une courbe synthétique est une moyenne pondérée ou non de résultats d'indicateurs synthétiques. Les courbes synthétiques sont caractérisées par :

– un code,
– une zone géographique appelée région,
– une pondération éventuelle,
– une fréquence, c'est-à-dire la périodicité de collecte des données,
– un secteur,
– une composition pondérée et qualifiée d'indicateurs,
– une méthode de calcul (désaisonalisation et/ou lissage).

Le programme est réalisé sur la plateforme .NET. Il en résulte une transposition du code du cache du Java vers le C#. Ce dernier langage étant très proche du Java, cela est tout à fait possible mais pas trivial. En effet, non seulement il existe des différences dans le langage même mais aussi dans les bibliothèques disponibles. Par exemple, en Java, une liste est déclarée par l'interface `List` et implémentée par la classe `ArrayList`. En C#, cette même liste est déclarée par l'interface `IList` et implémentée par la classe `List`. De plus, certaines méthodes ont la même signature mais pas le même effet.

L'application est contruite selon le modèle classique de 3-tiers, c'est-à-dire qu'elle est composée d'une base de données, d'un serveur central et de clients. Les traitements sont exclusivement effectués sur le serveur, les clients ne servant que d'interface avec l'utilisateur.

Deux problèmes classiques se posent : d'une part, de longs et coûteux calculs du coté serveur, et d'autre part, des transferts réseaux forcément lents.

11.1 Intégration

L'utilisation d'un modèle de type client-serveur avec un système de cache introduit le problème de la cohérence des données. Pour pallier ce problème, j'ai choisi de n'intégrer le cache que du coté serveur. Ce choix permet de simplifier le problème et est justifié par le fait que dans cette application, les calculs effectués sur le serveur sont bien plus longs que les transferts réseaux.

Du point de vue de la structure, l'application CS est fort complexe. Il existe beaucoup de dépendances entre les différentes méthodes qui la composent. Il est toutefois possible de structurer ces méthodes selon les données qu'elles manipulent. On peut ainsi distinguer quatre grands groupes :

- les méthodes qui manipulent des données stables, comme les secteurs et les régions
- les méthodes qui manipulent des lignes de synthèse
- les méthodes qui manipulent des indicateurs
- les méthodes qui manipulent des courbes

Les dépendances entre les groupes sont représentées par la figure 11.1 à la page 87.

Les courbes dépendent donc des indicateurs qui eux-mêmes dépendent des lignes de synthèse. Ces trois groupes dépendent des données stables que sont les régions et les secteurs. L'invalidation d'un des ces groupes engendrera l'invalidation des groupes qui en découlent.

Pour rappel, une invalidation consiste en la supression d'une donnée éventuellement stockée dans un cache lorsque sa valeur réelle est modifiée. Par exemple, si l'on ajoute des données à une série temporelle, comme une ligne de synthèse, toutes les séries qui en dérivent seront modifiées.

L'intégration se fait en deux étapes ; d'abord, il faut ajouter une annotation d'écriture à chaque méthode qui modifie les données en la paramètrant pour invalider le bon groupe. Ensuite, il faut déterminer quelles sont les méthodes de lecture les plus employées et y ajouter une annotation de lecture de nouveau associée au groupe correct.

11.2 Performances

Ce sont les performances globales de l'application, et non une méthode bien présise qui sont étudiées, au vu de la longueur de la liste des méthodes cachées.

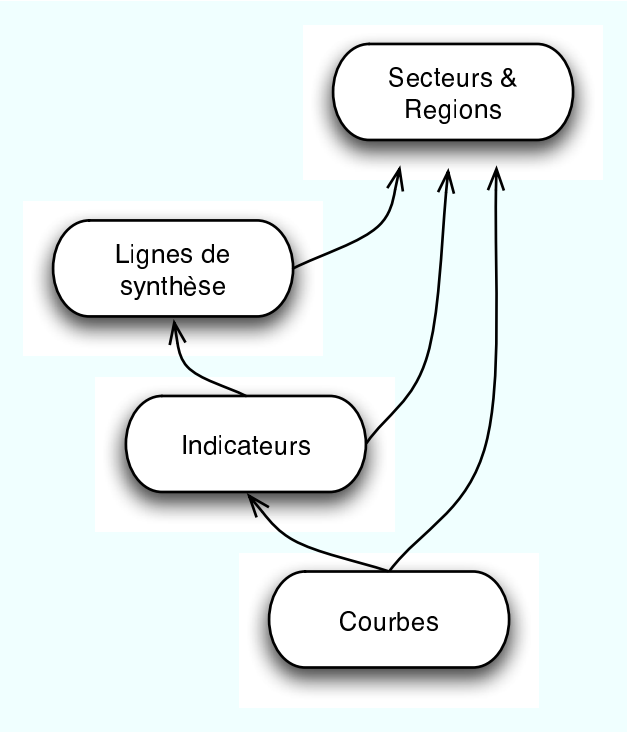

FIG. 11.1: Diagramme de dépendances entre les groupes de CS

Les méthodes sont assez complexes et leurs résolutions longues, certaines durant près de 800ms. De plus, les opérations de consultation sont bien plus nombreuses que les opérations de modification : il en résulte que le cache devrait être particulièrement efficace.

Le tableau suivant (tab. 11.1, p 88) montre les performances d'un jeu de tests répété quatre fois en modifiant la taille du cache (0, 10, 100, 1000). La méthode d'éviction choisie est la LRU.

Les temps d'exécution moyen et total diminuent avec l'augmentation de la taille du cache. De même, le pourcentage d'éléments valides récupérés dans le cache augmente. Toutefois, cette augmentation n'est pas proportionnelle à la taille du cache.

Pour sa part, le nombre d'appels diminue alors que le jeu de tests est identique. Ceci est dû au fait que certaines méthodes sont imbriquées. La mise en cache du résultat d'une méthode a pour conséquence que les méthodes qu'elle utilise ne sont plus appelées tant que le résultat est présent dans le cache.

Par ailleurs, l'analyse approfondie des logs fait apparaître un problème

Type de cache	Durée moyenne	Durée totale	% Valide	# Appels
Mémoire (0)	30 ms	37 s	0 %	1230
Mémoire (10)	24 ms	24 s	10 %	964
Mémoire (100)	21 ms	14 s	53 %	663
Mémoire (1000)	22 ms	12 s	62 %	546

TAB. 11.1: Impact de la taille du cache sur les performances de CS

de taille : la précision de la mesure du temps en .NET est plus faible qu'en Java. Cette précision est de l'ordre de 100 nanosesondes. Il en resulte que tous les appels cachés ou pas dont la durée d'exécution est inférieure à ce temps semblent nuls et faussent donc les résultats.

De plus, les requêtes lancées sur la base de données transitent par le réseau et leurs durées d'exécution sont donc dépendantes de celui-ci.

Ces problèmes de précision et de variabilité du réseau expliquent la légère augmentation de durée moyenne entre la configuration 100 et la configuration 1000. Les résultats obtenus sont donc dépendants du jeu de tests. Malgré tout, un cache de taille important reste plus performant.

Un autre aspect de la performance d'une application est la perception de sa rapidité par l'utilisateur. Elle n'est malheureusement pas quantifiable. L'expérience acquise montre toutefois une amélioration de la réactivité globale de l'application lors de l'activation du cache.

11.3 Constatations

L'utilisation de la programmation orientée aspect rend le débogage plus difficile. Il n'est en effet pas évident de déterminer si une erreur provient du code métier, de l'implémentation du cache ou alors d'un oubli dans les annotations. Pour limiter ce problème, il est indispensable de faire en sorte que l'utilisation du caching soit transparente et donc que l'application puisse fonctionner sans cache.

En intégrant tardivement et de façon transparente le mbc, il est possible d'éviter de torturer un code métier déjà de base fort complexe et de se consacrer entièrement aux fonctionnalités en laissant dans un premier temps les performances de côté.

Par ailleurs, en ciblant correctement les méthodes à cacher, les performances et le temps de réponse de l'application ont été grandement améliorés.

Enfin, cette application prouve que les théories développées dans ce mémoire ne sont pas limitées au langage Java mais sont transposables à d'autres langages. Par contre, il faut bien reconnaître que le Java dispose d'un plus grand choix de librairies, souvent de meilleure qualité et plus matures.

Conclusion

L'optimisation d'applications par caching peut être résumée en la création, à la demande, d'informations, avec un stockage temporaire en vue d'une réutilisation ultérieure éventuelle. En ce sens, Le caching est une combinaison idéale entre un fonctionnement statique et dynamique d'une application.

Le secret du succès du caching réside dans sa gestion automatique de la mémoire grâce, notamment, à des stratégies d'éviction. Le remplissage du cache est réalisé de manière transparente suivant les besoins des utilisateurs.

En outre, le caching permet de dépasser les limites de la programmation traditionnelle en augmentant les capacités de partage d'informations. En effet, une information n'est plus limitée à la classe qui l'a produite, mais peut être distribuée entre des classes, des processus et même des serveurs différents comme dans des fermes de serveurs web qui partagent entre elles les sessions des utilisateurs.

Utiliser du caching revient surtout à déterminer quand utiliser le caching et quand ne pas l'utiliser ; d'une part, il existe des techniques mieux adaptées à certains problèmes ou simplement mieux maîtrisées, et d'autre part, tous les algorithmes ne sont pas forcément cachables.

La deuxième grande difficulté du caching vient de son intégration dans le code métier. Cette difficulté est pourtant évitable en suivant une méthode simple en quatres étapes. Il suffit donc de :

- Développer sans cache dans un premier temps. Le caching est avant tout une technique d'optimisation. Elle n'est donc employée qu'en fin de développement, lorsque le code métier est fonctionnellement correct.
- Cibler les portions de code à optimiser. Seuls les goulots d'étranglements d'une application méritent un remaniement. Un développeur est souvent amené à devoir optimiser son temps et donc à se fixer des priorités.
- Choisir une solution existante. L'implémentation d'une solution de caching robuste et performante n'est pas trivial. Comme pour les bases de

données, il existe de très bonnes bibliothèques open-source librement téléchargeables sur Internet.
– Intégrer le caching de préférence par méta-programmation. Cette technique permet de séparer au mieux le code métier du code de gestion du cache. Le code résultant en est d'autant plus maintenable et extensible. De plus, les annotations, la version Java de la méta-programmation, permettent une publication simple de la présence de caching, que ce soit de manière programmatique ou dans la documentation javadoc.

Reste le problème de la gestion de la cohérence entre les données cachées et les données réelles. Dans la plupart des cas, cette gestion n'est pas automatisable et demeure donc à la charge du programmeur.

Le caching par méta-programmation est sans doute la technique ayant le plus de potentiel de développement. En effet, il est possible d'améliorer grandement l'API mbc proposée dans le chapitre consacré au caching dynamique de méthodes.

L'API mbc a été conçue pour être la plus généraliste possible. Cette qualité est parfois un défaut car elle n'est pas forcément adaptée à des problèmes particuliers comme, par exemple, des requêtes SQL.

En sus, la définition des dépendances entre les méthodes est réalisée manuellement. Une approche semi-automatisée serait très utile pour les programmeurs et pourrait par exemple être réalisée à l'aide d'outils d'analyse de code.

Par ailleurs, la mise en pratique des concepts développés dans ce mémoire prouve que les performances sont au rendez-vous. Le temps d'exécution de l'application CS est ainsi divisé par deux ou trois grâce à l'ajout d'un cache.

Toutefois, l'augmentation des performances n'est pas proportionnelle à la taille mémoire allouée au cache. Le profilage joue ici un rôle important car il permet de déterminer les paramètres optimaux qui assurent un équilibre entre rapidité et espace mémoire requis.

Malheureusement, il est assez difficile de créer un jeu de test qui reflète tous les types d'utilisation. Les performances que l'on obtient avec un cache peuvent être variables et difficiles à mesurer. Il en résulte que le caching n'est pas un remède miracle et son efficacité est liée à un bon design applicatif.

Enfin, le caching n'est pas une technique limitée au langage Java mais est transposable dans d'autres langages. L'avantage du Java se situe essentiellement au niveau de la disponibilité de bibliothèques open-source de grande qualité sur Internet.

Au final, le caching constitue une technique solide et sûre pour augmenter visiblement, et c'est là le plus important pour l'utilisateur, la vivacité d'une application. Et bien que cette solution ne soit pas neuve, il n'en demeure pas moins que le futur nous réserve sans nul doute de nouvelles améliorations.

BIBLIOGRAPHIE

ADAMS, D. (1979). *The HitchHiker's Guide to The Galaxy*. Pan Macmillian, London.

AWAIS, R. et BLAIR, L. (2001). Aspect-oriented programming and separation of concerns. http://www.comp.lancs.ac.uk/computing/users/marash/aopws2001/.

BORDERIE, X. (2006). Expliquez-moi la programmation orientée aspect. http://developpeur.journaldunet.com/tutoriel/theo/060421-paradigme-programmation-orientee-aspect.shtml. version du 26 avril 2006, consulté le 6 septembre 2006.

BOUCHENAK, S., COX, A., DROPSHO, S., MITTAL, S. et ZWAENEPOEL, W. (2005). Aop-based caching of dynamic web content : Experience with j2ee applications. Mémoire de D.E.A., Institut National de Recherche en Informatique et en Automatique, France.

BRIANGOETZ (2005). Interest group for jsr 107 - java caching api. https://jsr-107-interest.dev.java.net/. version du 19 janvier 2005, consulté le 16 juin 2006.

DAVISON, B. D. (2006). Web caching overview. http://www.web-caching.com/welcome.html. version du 10 mars 2006, consulté le 18 mai 2006.

ECKEL, B. (2006). Passage et retour d'objets. http://java.developpez.com/livres/penserenjava/?chap=17&page=0. version n/a, consulté le 16 juin 2006, chapitre du livre Thinking in Java traduit par Jérome Quelin.

FOWLER, M. (2004). *Refactoring - Improving the Design of Existing Code*, chapitre Refactoring and Performance, pages 69–70. Addison Wesley, Boston.

GAMMA, E., HELM, R., JOHNSON, R. et VLISSIDES, J. (1998). *Design Patterns : Elements of Reusable Object-Oriented Software*, chapitre Abstract Factory. Addison Wesley, Boston.

GEARY, D. (2001). Decorate your java code. http://www.javaworld.com/javaworld/jw-12-2001-designpatterns_p.html/. version du 14 décembre 2001, consulté le 14 décembre 2005.

GRAND, M. (2003). Pattern summaries : Cache management. http://www.developer.com/java/other/print.php/630481. version du 2 octobre 2003, consulté le 23 novembre 2005.

JAVA-SOURCE.NET (2006a). Open source cache solutions in java. http://java-source.net/open-source/cache-solutions. version n/a, consulté le 20 juin 2006.

JAVA-SOURCE.NET (2006b). Open source profilers in java. http://java-source.net/open-source/profilers. version n/a, consulté le 20 juin 2006.

JAVA.SUN.COM (2006). Java se real-time. http://java.sun.com/javase/technologies/realtime.jsp. version du 25 septembre 2006, consulté le 28 septembre 2006.

JBOSS (2006). Clustered caches. http://labs.jboss.com/file-access/default/members/jbosscache/freezone/docs/1.3.0.SP1/TreeCache/en/html/replication.html. version du 31 mars 2006, consulté le 20 juin 2006.

MCLAUGHLIN, B. et FLANAGAN, D. (2004). *Java 1.5 Tiger - A Developer's Notebook*, chapitre Annotations, pages 96–98 et 101–106. O'Reilly, Sebastopol.

PFEIFER, D. et JAKSCHITSCH, H. (2003). Method-based caching in multi-tiered server applications. Mémoire de D.E.A., Institute for Program Structures and Data Organisation, Universität Karlsruhe, Germany.

ROUX, L. (2004). J2se 5.0 tiger. http://lroux.developpez.com/article/java/tiger/?page=page_4. version du 17 août 2004, consulté le 21 juin 2006.

RUIZ, A. (2006). Declarative caching services for spring. http://dev2dev.bea.com/lpt/a/495. version du 19 mai 2006, consulté le 11 juillet 2006.

SCHOBBENS, P.-Y. (2002). *Méthodes de Programmation*, chapitre Mémoïsation, pages 111–118. FUNDP, Namur.

SHIRAZI, J. (2000). *Java - Performance Tuning*, chapitre When to Optimize, pages 357–359. O'Reilly, Sebastopol.

SMUTS, A. (2005). Jcs and jcache (jsr-107). http://jakarta.apache.org/turbine/jcs/JCSandJCACHE.html. version du 27 janvier 2005, consulté le 6 juillet 2006.

VIEL, N. (2005). Programmation orientée aspect : présentation et utilisation en c#. http://www.supinfo-projects.com/fr/2005/aop_overview_fr/1/. version du 17 mai 2005, consulté le 6 septembre 2006.

WHITE, T. (2003). Memoization in java using dynamic proxy classes. http://www.onjava.com/lpt/a/4088. version du 20 août 2003, consulté le 23 novembre 2005.

WIKIPEDIA.ORG (2005). Memoization. http://en.wikipedia.org/wiki/Memoization. version du 5 décembre 2005, consulté le 8 décembre 2005.

ZHICHENG-LI, J. (2006). Validate java ee annotations with annotation processors. http://today.java.net/pub/a/today/2006/06/29/validate-java-ee-annotations-with-annotation-processors.html. version du 29 juin 2006, consulté le 18 octobre 2006.